University of Hamburg
Department of Informatics
Databases and Information Systems

Master Thesis

Distributed Cache-Aware Transactions for Polyglot Persistence

---

Primary supervisor:	Prof. Dr.-Ing Norbert Ritter
Secondary supervisor:	Dr. Fabian Panse

Submitted by:

Erik Witt
Matrikelnr.: 6204671
Wasserstraße 19
25337 Elmshorn
0witt@informatik.uni-hamburg.de

Contents

List of Figures

1 Introduction

1.1 Motivation

With the availability of fast internet access and the wide adoption of mobile devices, web-based applications have been gaining immense popularity over the last years. The resulting huge potential user base and the constantly growing competition increase the importance of requirements like scalability, availability and low latency for business success.

Modern NoSQL systems meet these requirements by sacrificing consistency guarantees as well as features of traditional relational databases and are thus frequently used in large web applications. This system bias, however, conflicts with the need of application developers for rich feature sets and consistency guarantees that help them achieve a short time-to-market as well as an intuitive and consistent user experience.

The Backend-as-a-Service Orestes targets this field of data-driven web applications and mobile apps by offering easy to use backend modules through a client-side API while ensuring scalability, availability and especially low latency based on NoSQL and web caching technology. The offered functionality includes modules for data storage, queries, access control and user authentication among others.

While Orestes provides rich and tunable consistency guarantees on object level, transactions grouping multiple operations together have not been supported so far, due to the lack of support in the underlying NoSQL systems. Not having transactional access complicates application development and even rules out use cases with high consistency requirements. With the integration of transactions into Orestes, developers will be able to choose how to most efficiently access their data and when to use transactions, which handle concurrency and fault tolerance for them and enable several use cases of modern web applications based on collaborative and interactive operational patterns.

Therefore, the goal of this study is to design, implement and evaluate a concept for distributed transactions in the Orestes backend-as-a-service that matches the requirements of modern, data-driven web applications by adopting the Orestes performance properties of scalability, availability and low latency.

1.2 Structure of the Thesis

Chapter 2 describes the architecture and performance properties of Orestes and identifies requirements for the transaction concept. Chapter 3 introduces the theory on (distributed) transactions with a focus on concurrency control mechanisms and their suitability with respect to the identified requirements. The actual transaction concept is presented in Chapter 4 including details of its implementation, the recovery mechanism and an overview of its performance properties. The following Chapters 5 and 6 present optimizations to the transaction concept for write operations and read-only transactions respectively. The evaluation in terms of conflict rates and latency benefit of the concept as well as throughput and scalability of the implementation is given in Chapter 7 together with a real-world application example to showcase the usability. Finally, Chapter 8 shows an overview of related work in distributed transaction research before Chapter 9 concludes on the results of this study.

2 Orestes

Orestes is a Backend-as-a-Service (BaaS) based on research of the Information Systems Group at the University of Hamburg [GBR14, GFW+14, GSW+15, SGR15, GSW+16]. The outstanding feature of Orestes is that it solves the problem of high network latencies in cloud-based applications. This problem is tackled by an advanced caching mechanism in front of a distributed, database-independent backend that provides elastic scalability and tunable consistency configurations on a per-object and per-operation basis.

This chapter presents the web caching solution of Orestes along with an architectural overview and an assessment of its performance properties. The requirements for the integration of a transaction concept will be inferred from this analysis.

2.1 High Network Latencies

The load time of websites and mobile apps is one of the most important metrics of web performance, as numerous studies have shown its high impact on traffic, conversion rates and revenue [Gri13, p. 171] [Bru09, Hof09]. Amazon for example found that $100\,ms$ of additional page load time decreases their revenue by 1%, which translates to 1.07 billion dollars of lost revenue annually [Hof09].

With over 100 requests on average to load a full web page, consisting mostly of database lookups and small requests, page load time is almost exclusively governed by round-trip latency [Arc16]. For TCP-based communication, round-trip latency is the time between issuing a request to receiving a response [Gri13, p. 3f]. While round-trip latency depends on many components of the underlying network, it is ultimately bounded by the distance from clients to accessed data. The only way to effectively and generally reduce this latency is to bring data closer to the client. [Gri13, p. 6, 165ff]

For this purpose the HTTP protocol integrates an expiration-based caching model where caches act as proxies (forward or backward proxies) and store the response on the way to the client [Wes01]. Cached responses can be directly returned for subsequent requests and are stored for a fixed time span called *time to live (TTL)*, unless a revalidation request is issued to renew the response [Gro99]. There are several of these caches forming a cache hierarchy. The most relevant ones are depicted in Figure 2.1.

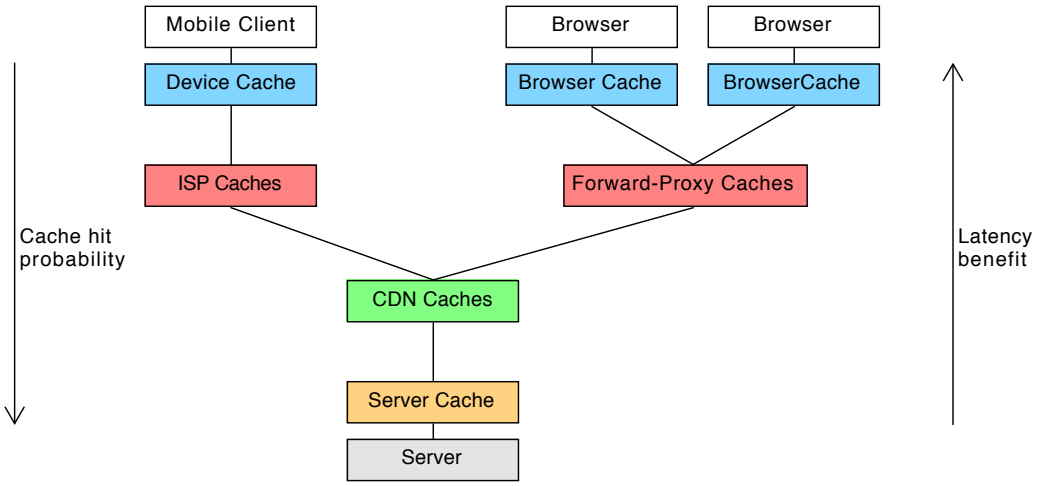


Figure 2.1: Hierarchy of HTTP-based caches

- **Client Cache**: Every client has an integrated client cache, either as a browser or device cache. This cache has the highest latency benefit as cache hits return responses instantly, but also the lowest probability of a cache hit as it is only filled with responses sent to this client. [Wes01, p. 15]

- **Forward-Proxy Caches**: A Forward-Proxy cache can be deployed in the client's network, a company's network for example [GTS+02, p. 168f]. This cache is shared by all clients in the network and thus, has a higher cache hit rate than the client cache while retaining a high latency benefit. The cache hit rate improves with the number of user, assuming they request overlapping resources.

- **ISP Caches**: Clients that have the same internet service provider (ISP) also share the set of caches hosted at their ISP. These caches are still relatively close to the client and have a higher cache hit rate because they are filled with responses to any client of the ISP. [Wes01, p. 15f]

- **CDN Caches**: At the next level are content delivery networks (CDNs) that deploy globally distributed caches. The global distribution makes these caches very efficient as they are close to most clients and have a high cache hit rate because they are filled with responses to many clients. [Gri13, p. 7]

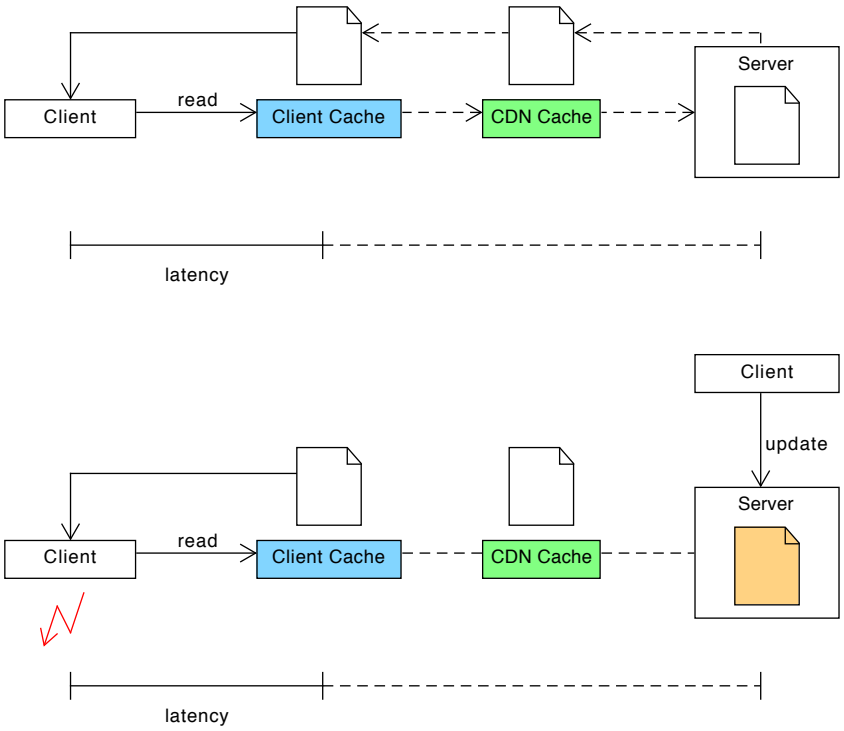


Figure 2.2: Caching of static (above) and dynamic data (below)

- **Server Cache**: Server caches (surrogate caches) at the last level are deployed in front of or within the application server. These caches do not reduce the distance to the client but rather free the backend from processing the request. [Wes01, p. 16f]

This HTTP caching infrastructure is widely used for static resources, but the expiration-based caching model is not well-suited to serve non-static data like database records.

The problem with database records is that after an update, caches continue to serve their old responses until the TTL expires or a revalidation request is issued. The first diagram of Figure 2.2 shows the correct cache behavior, where the response of the first request (dashed lines) is cached in both the CDN and the client cache and the second request is served by the first cache (solid lines). The second diagram shows the problem that arises when the record is updated by another client. The caches continue to serve the old (also called stale) records and thus present outdated data to the client. This behavior is inadequate for many applications.

Geo-Replication

The common approach to reduce latency for database reads is *Geo-Replication*, i.e. building a globally distributed system over multiple data centers and replicating the data over these locations [LFKA13]. Geo-Replication is quite complex as well as expensive and therefore not suitable for smaller web applications. The cache coherence mechanism of Orestes solves the problem of caching non-static data and thus is able to leverage the existing global web caching infrastructure of the HTTP protocol to serve database objects with significantly reduced latency. The next section describes the caching mechanism in detail.

2.2 The Caching Architecture

Orestes distinguishes between *invalidation-* and *expiration-based* caches [GSW+15]. Invalidation-based caches are those of the hierarchy that can be invalidated by the server, i.e. the server can notify invalidation-based caches to drop cached entries of updated records. Expiration-based caches are those that can only be revalidated by the client. CDN and server caches are invalidation-based whereas client, forward-proxy and ISP caches are expiration-based.

In order to increase the consistency of the cached data, Orestes deploys two separate strategies for invalidation- and expiration-based caches. The strategy for invalidation-based caches is to invalidate them once an update is received. The strategy for expiration-based caches is more complex because those caches can only be revalidated by clients. The client in turn can only efficiently revalidate cache entries if stale objects are known. To this end, the client periodically requests the set of recently updated records, which is sent in a highly compressed, probabilistic set called a Bloom filter [Blo70]. The Bloom filter encodes object ids using hash functions and answers membership tests with a small false positive rate. The two invalidation strategies are shown in Figure 2.3 and work together as follows.

If an update to a record (1) is received by a server, an invalidation request (2) for that record is sent to all invalidation-based caches and the record is added to the set of recently updated records maintained as a Bloom filter at the server side[1]. A copy of the Bloom filter is periodically requested (3) by the client.

If a client loads a record, the API checks if its id is contained in the client's Bloom filter (1). If it is contained i.e. the record was updated recently, a revalidation request (2) is issued, which loads the newest version and updates the caches. If the id is not contained,

[1] Records are removed from the set when their TTL expires.

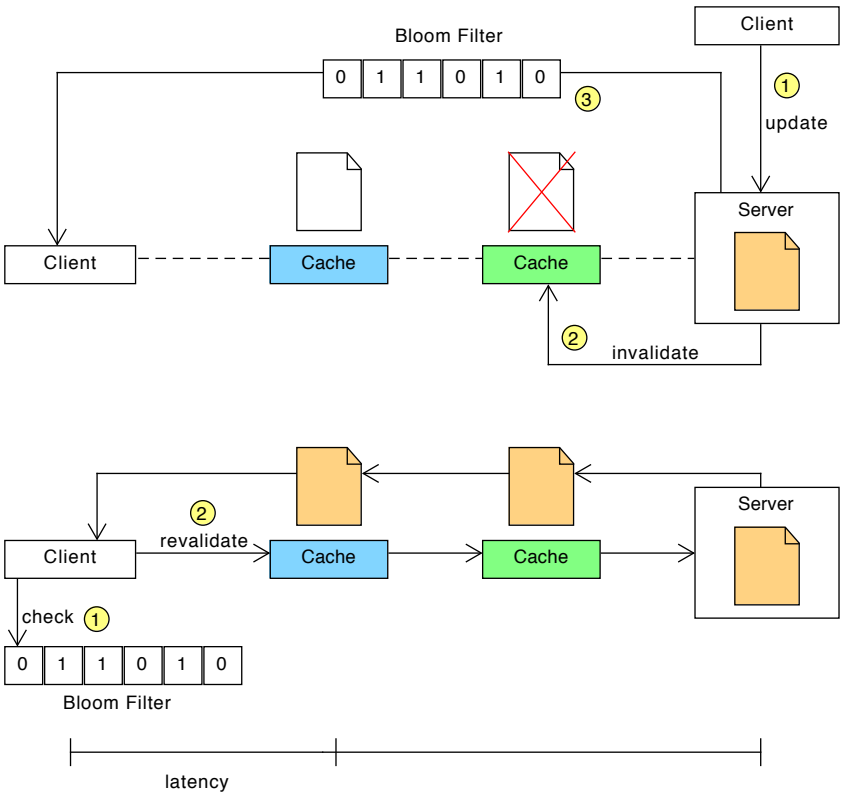


Figure 2.3: Invalidation (above) and revalidation (below) of caches

a normal request is issued, which may be answered by one of the caches. It is important to note that invalidation-based caches – especially the CDN – can answer revalidation requests from its cache instead of forwarding them to the server. Otherwise revalidation requests from every client would be forwarded although the cache has already been refreshed.

2.2.1 Consistency Strategies

Using the Bloom filter as a basis for revalidation is only one out of four strategy to load a record described in [GBR14]:

1. **Read Any (RA)**: Clients send simple requests, which may be answered by a cache, regardless of whether stale or not.

2. **Read Newest (RN)**: Clients send revalidation requests to receive the newest version.

3. **Bloom-Filter-Bounded staleness (BFB)**: Clients use the Bloom filter to revalidate recently updated records as described above.

4. **Transactional (TA)**: Clients use RA requests for reads which are validated at commit time. This strategy is not yet supported by Orestes for the lack of transactions. The strategy will, however, be part of the requirement analysis in Section 2.4.

The strategy RA has the highest latency benefit because the data is served from the nearest location where it is available. In turn the consistency of RA is bounded only by the TTL which is typically chosen high to increase cache hit rates.

The strategy RN requires a full round-trip for each request and thus has no latency benefit. With the additional latency compared to RA, RN achieves strong consistency[2].

The strategy BFB is a tradeoff between latency and consistency. Reads of recently updated records require a full round-trip, while others can be served by any cache. BFB guarantees a consistency bounded by the time the Bloom filter was loaded, i.e. "clients are guaranteed to see only object versions that are at least as recent as the database state by the time the Bloom filter was generated." [GBR14]. It can also be used to guarantee read-your-write, monotonic read and causal consistency [GSW+16]. BFB is the recommended option and the best tradeoff between latency.

"A consistency strategy can be chosen per operation, session, [...] or application and mixed according to the application's needs." [GBR14] This caching architecture is embedded in the BaaS Orestes to make it easily accessible for application developers. Like the caching mechanism, the BaaS architecture has a high influence on the transaction requirements and is therefore described in the next section.

2.3 Architectural Overview

2.3.1 Components

The main components of the Orestes architecture as described in [GBR14] are the *REST API*, the *caching architecture*, the *REST servers* (called *Orestes-Server*) and the *underlying database system*. Figure 2.4 shows an overview of these components.

- **REST API**: Orestes exposes its database and other backend functionality through a REST API to provide a uniform, stateless interface with well-defined semantics and

[2] Assuming the underlying database system provides strong consistency.

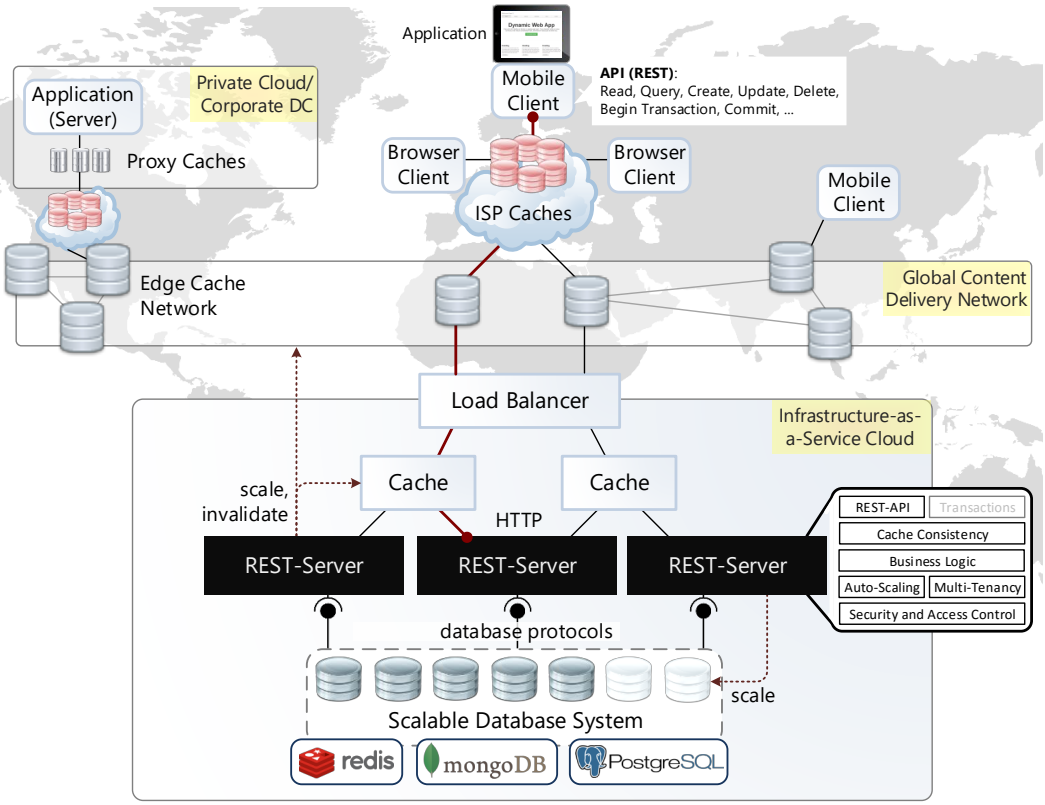


Figure 2.4: Orestes architecture overview based on [GBR14]

to leverage the existing HTTP caching infrastructure. The application developer uses the REST API or a native SDK[3] to load database records via GET requests.

- **Web Caching**: The request can either be answered by one of the caches resulting in highly reduced latency as described in Section 2.2 or it is forwarded to one of the REST servers. The caching mechanism is fully transparent and performed in the client API and on HTTP level. The CDN caches in front of the servers also serve as load balancers and forward the request to one of the REST servers according to their policy.

- **REST Server**: Because REST servers are stateless, i.e. they do not rely on any state from previous requests, the load balancer is free to choose any REST server.

[3]Currently native SDKs for JavaScript and Java are available.

The REST server is responsible for the backend functionality including schema management, cache coherence, authentication, security, access control and business logic as well as persistence.

- **Scalable Database System**: The REST server supports polyglot persistence and uses an underlying scalable database system. Orestes supports different databases that can be plugged into the system to adapt to different use cases[4]. Depending on the used database, Orestes offers different SLAs (Service Level Agreements) and extends the REST API with database specific queries for example. The long term vision for the database system is a polyglot persistence mediator described in [SGR15], which decides automatically which database system to use for any given part of the schema, based on SLAs specified by the application developer.

The architecture and implementation of Orestes are designed to minimize load times while offering rich consistency guarantees. The next section explains the influence of the architectural components on key performance properties of large-scale web applications and assesses the implementation of Orestes.

2.3.2 Performance Properties

- **Low Latency**: Minimized load times are achieved by the caching architecture which reduces round-trip latency and eliminates processing time in the backend. Write operations on the other hand are always sent to the server and have no benefit regarding round-trip latency. Therefore the main target of Orestes are "read-intensive, latency-sensitive workloads common for most web applications" [GBR14].

- **Consistency**: With the three load strategies RA, RN and BFB (and the fourth one TA to come), rich consistency guarantees are available that can be traded against latency. Especially BFB is a well suited tradeoff for common web applications.

- **Elastic Read Scalability**: For reads the architecture scales in two ways. First, the distributed web caching infrastructure is naturally highly scalable and even grows with each client (by the client cache). Furthermore, the cache hit rates increase with the number of reads, making caching even more efficient. Especially *flash crowds* [VP03] – common for web applications – are easily handled by the caching architecture. Second, reads that are not answered by caches are load-balanced over the stateless REST servers, which can elastically be added or removed. Assuming

[4] Currently Redis and MongoDB are used. Support for Cassandra, Elasticsearch, PostgreSQL and DynamoDB is in development.

a scalable underlying database, this achieves elastic read scalability for operations that reach the server.

- **Elastic Write Scalability**: Writes are always load-balanced across the REST servers that can elastically be added or removed. Assuming a partitioned underlying database, this achieves elastic write scalability.

- **Availability and Fault Tolerance**: The availability of the REST servers is achieved by redundancy (multiple REST servers registered at the load balancer). The availability of the database depends on its configuration as consistency, availability and partition tolerance can not be guaranteed by any distributed system, as proven by the CAP theorem [GL02]. Thus a database system with strong consistency may be unavailable during a network partition. An available system on the other hand does not guarantee strong consistency. For fault tolerance the main strategy is to automatically restart failing REST servers, Node servers (that execute client code) and database nodes. Failing components are identified using health checks and handled by the cluster manager and load-balancer.

The assessment of the key performance properties shows conceptual evidence for the strong performance capabilities. Furthermore the combination of tunable consistency and latency with the interchangeable underlying database system make Orestes extremely versatile as a BaaS. Nevertheless Orestes lacks one fundamental feature needed by many applications. Besides the rich consistency guarantees for single read operations, properties for the execution of multiple related operations have to be guaranteed. Environments that guarantee properties for the execution of related database operations are commonly known as *transactions*.

The following section summarizes the requirements that the implementation of a transaction concept on the Orestes architecture should fulfill.

2.4 Requirements for Transactions

The purpose of transactions and this the supreme requirement is to ensure semantics for groups of database operations that are strong enough to handle all relevant use cases. The integration of the transaction concept into Orestes adds further requirements.

To meet Orestes's focus on load times, transactions have to induce *low latency* on all their related operations. Additionally the properties *elastic scalability*, *availability* and *fault tolerance* have to be carried over from Orestes. Furthermore transactions need to

be *database independent* to be used in combination with polyglot persistence. Lastly, the transaction concept has to be *widely applicable* to satisfy most common use cases.

The goal of this study is to design, implement and evaluate a transaction concept for Orestes that meets these requirements. The next chapter will cover the relevant theory on distributed transactions and lead to the first design decisions.

3 Distributed Transactions

3.1 ACID Guarantees

A major problem in data persistence is to keep data consistent even in the presence of highly concurrent data access and despite all sorts of failures [WV02, p. 4]. The key benefit of transactions is that they free the application programmer from these issues of concurrency and failures. To this end, transactions guarantee certain properties for all requests inside their dynamic boundaries. These guarantees are known as the ACID properties and consist of *atomicity*, *consistency*, *isolation* and *durability* [WV02, p. 22ff]. In essence, a transaction is a sequence of operations transforming the database from one consistent state to another. [WV02, p. 24f]

This section will present the ACID properties in detail and illustrate them by an example transaction. For this example consider two bank accounts A and B with a certain amount of money. The transaction transfers money by withdrawing $100\,\$$ from account A and deposing it to account B.

Atomicity

The property atomicity means, as Weikum and Vossen [WV02, p. 23f] state, that "From the client's and the application program's point of view, a transaction is executed completely or not at all". Thus, in the absence of failures or conflicts the transaction is expected to commit, resulting in a database state where all operations are applied successfully. If failures or conflicts occur during the transaction, the processing will be aborted and no operation will be applied persistently. An aborted transaction behaves as if it has never been invoked at all.

For the example this means that it will never happen that the withdrawal from account A is executed without the deposit to account B or vice versa. Either both operations are executed or none of them. Thus, atomicity protects money from entering or leaving the system due to partially applied transactions.

Consistency

The property consistency, explained in [WV02, p. 24], means that all consistency constraints defined for the underlying data are preserved by a transaction. Thus, at the commit of a transaction all constraints must be satisfied even though during the transaction execution intermediate states may violate them.

Since most consistency constraints are individually defined by the client, this guarantee cannot be fully automated. The client has to make sure that a transaction produces only consistent database states, assuming a consistent state at transaction begin. In turn the transaction processing ensures the constraints for the whole system. These transactions are called *logically consistent* [SSS15, p. 10]. In addition to designing logically consistent transactions, some constraints can usually be specified in the database management system (DBMS) and automatically be validated before the transaction's commit. If the validation fails, the transaction will automatically be aborted.

In the example, an obvious constraint would be that the overall balance of all accounts stays the same. The example transaction is logically consistent with respect to this constraint because the same amount is withdrawn and deposited and no money enters or leaves the system. Another constraint that could be automatically ensured by the transaction processing would be that no account is allowed to have a negative balance. Accordingly, the transfer of $100\,\$$ from an account with a balance of $50\,\$$ would be aborted to ensure consistency.

Isolation

Isolation is the property that deals with concurrency in the execution of transactions and prevents the anomalies that can arise from it. According to [WV02, p. 24f] "[a] transaction is isolated from other transactions, in the sense that each transaction behaves as if it were operating alone with all resources to itself". Thus, two concurrent transactions do not affect each other.

Consider for example that the withdraw from account A actually consists of separate operations, namely reading the current amount, subtracting $100\,\$$ and then writing the new amount. If two transactions t_1 and t_2 both withdraw from account A, they both read the current amount of A (say $200\,\$$) and subtract their transfer amounts (both $100\,\$$). If both write back the calculated amount, one of the new values will be overridden and the amount of A will be $100\,\$$ instead of $0\,\$$. This *lost update* is one of the anomalies that can result from concurrent transaction processing and has to be prevented by isolation.

Property	Definition	Implementation
Atomicity	The transaction is applied completely or not at all.	Recovery and rollbacks
Consistency	A transaction transfers the database from one consistent state to another.	Logically consistent transactions with atomicity and isolation plus consistency constraint checking.
Isolation	Transactions that are processed concurrently do not affect each other.	Concurrency control
Durability	Changes of committed transactions are persistent and guaranteed to survive subsequent failures.	Logging and recovery

Table 3.1: ACID properties and their realization

Durability

The property durability means that data written by a successfully committed transaction is persisted and "guaranteed to survive subsequent software or hardware failures." [WV02, p. 25].

This means for example that the successful money transfer is durable and will not be undone by any future software or hardware failure.

Centralized Implementation

Implementations of the ACID guarantees for centralized database systems have been well studied and some are widely used in practice. As Table 3.1 shows, atomicity can be achieved by rolling back failed transactions or recovering successful ones. Consistency is the combination of logically consistent transactions, atomicity (to avoid inconsistent intermediate states) and isolation (to avoid anomalies that violate consistency). Isolation is achieved by concurrency control mechanisms like locking or optimistic approaches. Durability is ensured by logging and recovery. [HR13, p. 393ff]

The following section discusses the additional challenges that arise for distributed database systems and the common approach to implement distributed transactions.

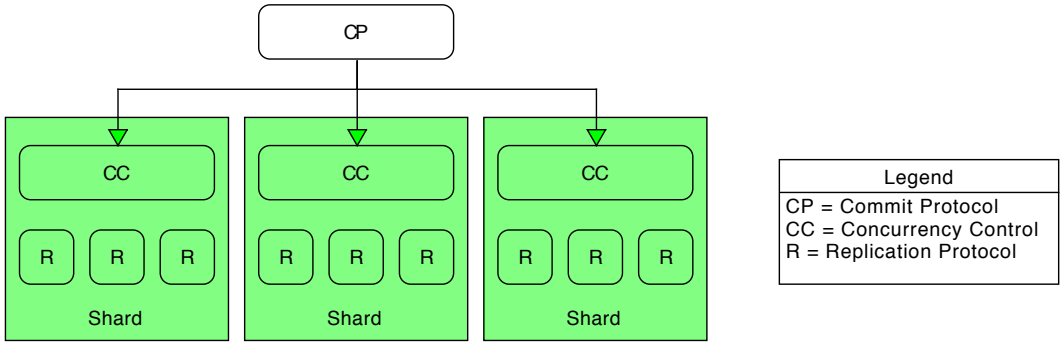


Figure 3.1: Common distributed transaction setup

3.2 Implementation Approaches

Distributed database systems "commonly partition data into shards for scalability and then replicate each shard for fault-tolerance and availability" [ZSS+15]. Incoming transactions are also distributed over the shards such that each shard handles all operations that access its data. Figure 3.1 shows the typical approach to distributed transactions on such systems according to [ZSS+15]:

CP The commit protocol ensures the atomicity of the transaction i.e. it ensures that each shard executes its part of the transaction successfully or not at all. Additionally the commit protocol ensures consistency either by checking conditions itself or by reacting to the consistency checks performed on the shards. If any shard aborts its part of the transaction, the commit protocol has to abort it on all other shards. In essence the protocol achieves a unanimous decision about the success of the transaction. The most prevalent commit protocol is the Two-Phase-Commit Protocol (2PC) [Lec09].

CC Concurrency control is performed at shard level for each part of the transaction separately. The protocol has to ensure that the execution on each shard leads to a globally correct isolation. The implementation typically uses locking or optimistic protocols.

R The replication protocol has to ensure the durability and consistency of the data written to the shards. While each replica uses logging and recovery for durability, the replication protocol handles the data transfer between the replicas.

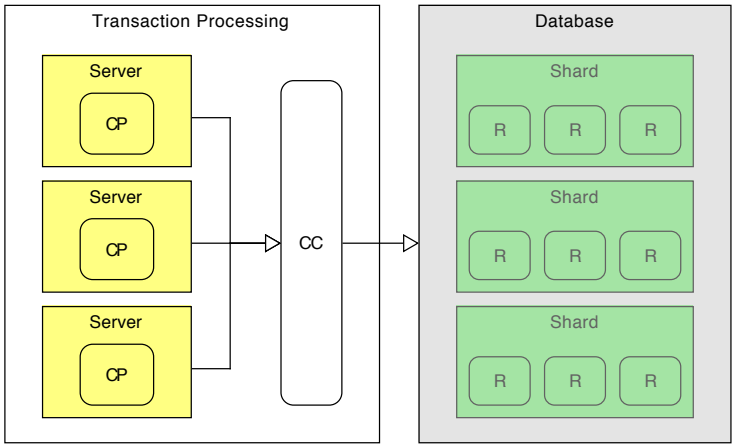


Figure 3.2: Possible polyglot persistence transaction setup

This common approach does not fit the requirements of this study because for polyglot persistence transactions the database system and therefore the partitioning and replication need to be separated from the transaction processing and thus can be thought of as a black box. Figure 3.2 shows a different approach that takes this separation into account and distributes transactions over multiple servers for transaction processing, instead of partitioning each transaction over the shards.

Besides the separation of the transaction processing and the database system, the two approaches differ in the way the commit handling and the concurrency control are applied. Because in the second approach each transaction is handled by one server, a local commit protocol is sufficient whereas in the first approach the commit decision of all shards must be combined by a distributed commit protocol. For concurrency control it is the other way around. The first approach can apply a local protocol on each shard because all operations that access the same object are assigned to that shard. The second approach in contrast needs to apply a distributed protocol to detect or prevent conflicting operations of transactions handled by different servers.

In summary, the commit protocol and the concurrency control are the main components for the implementation of distributed polyglot persistence transactions. Therefore, the next section discusses the different approaches to concurrency control and establishes a formal correctness criteria for concurrent transaction processing. The commit protocol is determined by the choice of the concurrency control mechanism and will be explained in Chapter 4. The recovery mechanism also depends on the concurrency control algorithm and will be discussed in Subsection 4.3.2.

3.3 Concurrency Control

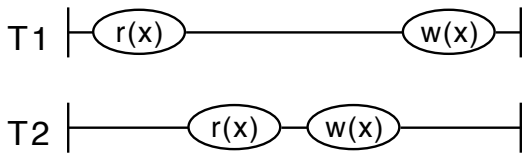
3.3.1 Anomalies

The concurrent execution of transactions can lead to anomalies that produce unintended database states and can violate consistency. This section will outline the common anomalies and illustrate them by the money transfer example from the previous section.

To reason about their concurrent execution, transactions are considered as a finite sequence of indivisible read $r(x)$ and write $w(x)$ operations on data objects. For a set of transactions a history describes the entanglement of the transactions' operations. A prefix of a history is called a schedule. [WV02, p. 44, 65f]

Lost Update

A lost update anomaly occurs if a transaction overrides a value based on a stale version of that value. The following schedule shows a generic example where transaction T_1 reads the value of object x and then writes back a new value. In the meantime transaction T_2 has written a value for x that T_1 did not read resulting in the loss of the update of T_2. [WV02, p. 62f]

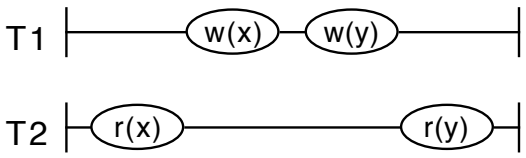


For the money transfer example the transactions T_1 and T_2 can be considered as two independent deposits to the same account. For the deposit both transactions read the current balance of the account, add their respective deposit amounts (say $100\,\$$) and write back the new balance for the account. The result of the schedule is that one of the deposits (and therefore $100\,\$$) is lost.

Fractured / Inconsistent Read

A fractured / inconsistent read anomaly occurs – as illustrated in the following schedule – if a transaction T_2 reads a value y written by another transaction T_1 along with a stale

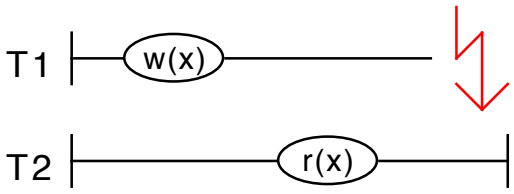
value x later written by the same transaction T_1. The non-repeatable read anomaly is a special case of a fractured read, where x and y are the same object. [WV02, p. 62f]



For the money transfer example, the transaction T_1 can be considered as the transfer of some amount (say $100\,\$$) from account x to y. T_2 reads both accounts to get the overall balance. The anomaly leads to an overall balance of the accounts that appears ($100\,\$$) higher than it actually is.

Dirty Read

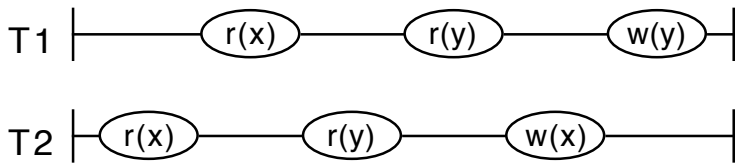
A dirty read anomaly occurs if a transaction reads a value from another transaction that gets aborted afterwards. The following schedule shows an example where T_2 reads the value of T_1 which is aborted afterwards.



For the money transfer example the transaction T_1 can be considered as a withdraw of some amount (say $100{,}000\,\$$) that is aborted after the database system validates the consistency constraint that the amount of an account must stay positive. Transaction T_2 reads the amount of the account – as in a bank account statement – and returns an inconsistent negative amount that is not allowed to exist.

Write Skew

A write skew anomaly occurs – as illustrated in the following schedule – if two transactions T_1 and T_2 concurrently read an overlapping data set ($\{x, y\}$) and perform disjoint writes (y in T_1 and x in T_2) afterwards. [CRF08]



The money transfer example illustrates why this can be a problem. As in an example in [FLO$^+$05], consider the consistency constraint that the added balance of account x and account y must be positive. Both transactions T_1 and T_2 first check that the added balance of x and y is high enough and then withdraw their amounts from x and y respectively. Each transaction on its own would preserve the constraint, but the concurrent schedule could violate it because the concurrent updates are not taken into consideration.

The write skew anomaly is often used to distinguish serializable isolation from the weaker veriant *snapshot isoltation (SI)*.

Others

There are other anomalies (see [Ady99]) that can occur due to concurrent transaction processing but the discussed examples sufficiently show that a formal correctness criteria for concurrent schedules is needed along with a concurrency control mechanism to enforce it. The next subsection presents the correctness criteria.

3.3.2 Serializability

The foundation of the correctness criteria for concurrent schedules is the following observation from [WV02, p. 72]: "Assuming that schedules should maintain the integrity of the underlying data and assuming each individual transaction is able to assure this, it is reasonable to conclude that serial histories are correct". A concurrent schedule is called serializable iff there exists a serial schedule that is equivalent in its result and its effect. Building on this foundation, serializability is the criteria used to define correctness.

There are multiple classes of serializable schedules that distinguish themselves by using different notions of equivalence, which lead to varying properties. In the following, the most important of these classes are presented.

Final State Serializability (FSR)

Two schedules are final state equivalent iff they produce the same final state for any given initial state. A schedule is in FSR iff there exists a serial schedule in its final state equivalence class. [WV02, p. 76ff]

FSR has two major drawbacks. First, only the effect of the transaction on the database and not its returned result is considered in the equivalence definition. Therefore FSR does not prevent inconsistent read anomalies. Second, the computation to decide whether or not a schedule is in FSR is NP complete [Set81] and therefore not efficiently decidable.

View Serializability (VSR)

VSR is a strict subset of FSR and additionally requires that schedules read the same values to be view equivalent. This means their transactions read the same data objects with the same values. [WV02, p. 83ff]

Taking results and effects into account VSR prevents the discussed anomalies[1] including inconsistent reads. While VSR is considered to be sufficient for consistent transaction management, it shares the second drawback with FSR: the containment decision is not efficiently decidable [Set81].

Conflict Serializability (CSR)

CSR again is a strict subset of VSR and uses a fairly different definition of equality, which is based on the notion of a conflict and is efficiently decidable.

A conflict is a temporally directed relationship between two operations. Two operations of different transactions are in conflict if both access the same object and at least one of them is a write operation. Two schedules are conflict equivalent iff they have the same operations and the same conflicts. [WV02, p. 93f]

The containment in this class is efficiently decidable using the conflict graph of the schedule. The conflict graph of a schedule S has a node n_i for each transaction $t_i \in S$ and an edge $n_i \rightarrow n_j$ for each conflict from one operation in t_i to one in t_j. A schedule is in CSR iff its conflict graph is acyclic. [WV02, p. 97] Figure 3.3 shows an example of a not conflict serializable schedule with the corresponding conflict graph.

[1] Dirty reads are not prevented in the serializability classes but handled separately.

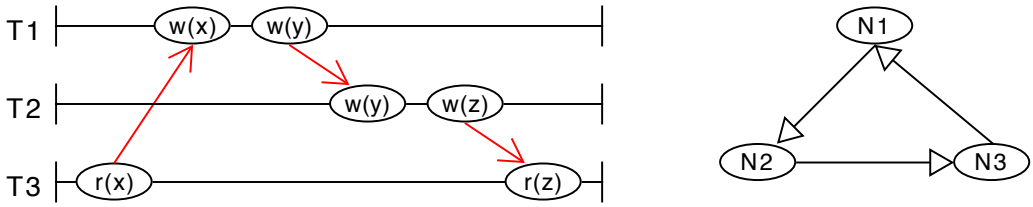


Figure 3.3: A non-serializable schedule with its corresponding conflict graph

Order Preserving Conflict Serializability (OCSR)

OCSR restricts CSR such that transactions that do not overlap in the schedule must appear in the same order in the equivalent serial schedule. Although CSR is sufficient for correctness and efficiently decidable, the restriction to OCSR makes sense if users want transactions to be executed in the order they are submitted and wait for transactions to commit before submitting the next one. [WV02, p. 101f]

Commit Order Preserving Conflict Serializability (COCSR)

COCSR restricts CSR such that if the conflict graph contains an edge from n_i to n_j i.e. there exists a conflict from t_i to t_j, then t_i must commit before t_j [WV02, p. 102f].

COCSR has an especially useful property for distributed environments. If all local partitions of a global schedule (on each shard for example) are in COCSR then the global schedule is conflict serializable. This follows from the global commit order and a theorem form [WV02, p. 678] which states that local conflict serializable schedules combined with a consistent ordering of the transactions (the commit order in COCSR) lead to a conflict serializable global schedule.

Conclusion

The serializability classes described in this section are strictly contained in each other as shown in Figure 3.4. The smaller classes offer richer properties, but in turn restrict concurrency and therefore performance and scalability more than the larger classes.

In the remainder of this study, CSR will serve as the correctness criteria for concurrent schedules. The next subsection describes several common concurrency control algorithms that enforce serializability and discusses their pros and cons for the use case of this study.

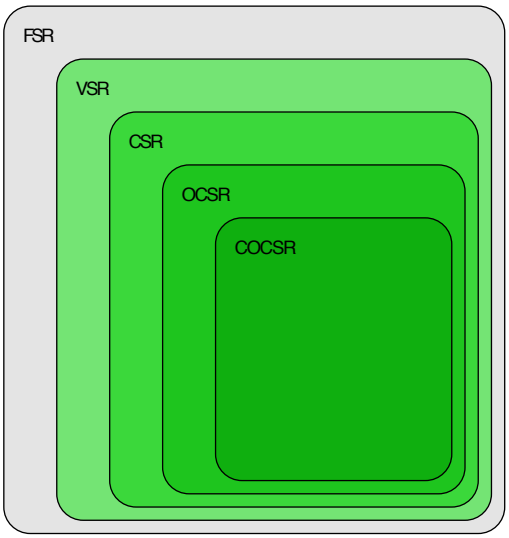


Figure 3.4: Serializability classes

3.3.3 Concurrency Control Algorithms

This section describes a selection of the most common concurrency control algorithms, which each fall into on of the following categories: locking pessimistic, non-locking pessimistic or optimistic.

Locking

Algorithms of this category use locks with different modes ensure serializability. To read a data object, a transaction has to acquire a shared lock (read lock). To write a data object an exclusive lock (write lock) is needed. As shown in Figure 3.5, two shared locks are compatible i.e. read access can be granted for several transactions simultaneously. An exclusive lock on the other hand can only be held by one transaction at a time, thus other transactions that request a lock for that object are blocked until the exclusive lock is released. The compatibility of locks corresponds to the definition of a conflict. [WV02, p. 130f]

The **two-phase locking protocol (2PL)** strictly separates the phase in which locks are acquired from the phase where locks are released. Thus, after a transaction has released the first lock it cannot acquire another one. Schedules produced by the 2PL protocol are

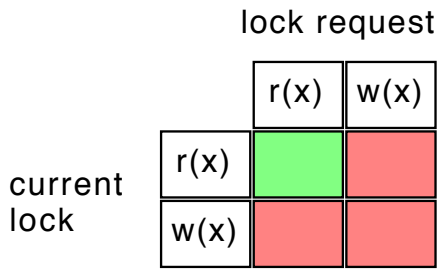


Figure 3.5: Lock mode compatibility

in OCSR. If all locks are held until commit, the protocol is called strong 2PL (SS2PL) and all produced schedules are in COCSR. [WV02, p. 134, 144]

2PL is the most common protocol in centralized database systems [WV02, p. 133]. One major advantage of 2PL is that conflicts of transactions are resolved by locking and do not lead to aborts – apart from deadlocks. Nevertheless its use in distributed systems has a number of drawbacks. The main problem is that lock requests can produce deadlocks where one transaction is waiting on another that is directly or transitively waiting for the first transaction to release locks. If not detected and resolved, deadlocks cause transactions to block indefinitely. “However, deadlock detection in distributed databases is considerably more difficult than in centralized ones” [WV02, p. 680]. The coordination needed for deadlock detection would decrease scalability.

A possible solution to the deadlock problem is to order lock requests (**Ordered 2PL**). If every transaction requests locks in the same order, deadlocks cannot occur. The main problem with this variant is that all objects to lock must be known beforehand.

The main drawbacks arise from the requirements of this study’s use case. To acquire the locks, each read and write operation of a transaction has to be sent to the server. Thus, reads cannot be served from caches and writes cannot be buffered at the client. Both restrictions would highly increase latency.

Non-Locking

The algorithms of this category ensure serializability without using locks. This section will present two algorithms that achieve this in different ways.

In the **timestamp ordering (TO)** algorithm a timestamp is drawn from a totally ordered domain for every transaction. The algorithm guarantees that if two operations

of different transactions are in conflict, they are executed in timestamp order. The implementation works as follows. At transaction begin a unique monotonically increasing timestamp is generated for that transaction and used for all its operations. An operation of a transaction is executed immediately iff no conflicting operation of another transaction with a higher timestamp has been executed before. If an operation cannot be executed the corresponding transaction is aborted. All produced schedules are in CSR. [WV02, p. 166f]

The main problem with TO is that compared to 2PL, many transactions are aborted. The abort probability increases with the age of the timestamp, which is problematic especially for long running transactions and for use cases with many concurrent transactions.

Furthermore, the implementation of TO in a distributed system limits the scalability in multiple ways. Firstly, the timestamp has to increase monotonically for each starting transaction, which necessitates a centralized timestamp oracle or a coordination intensive distributed algorithm. Secondly, for each operation it must be decided if it can be executed. Thus, information on the last reading and the last writing transaction must be loaded. This validation must be executed as a critical section for each object, as to prevent transactions from concurrently writing and validating objects.

Again, further drawbacks arise from the requirements of this study's use case. To validate and execute a transaction, the read and write operations have to be sent to the server which rules out caching and highly increases latency.

The **serialization graph testing (SGT)** algorithm uses the conflict graph of all transactions to validate that the applied schedule remains serializable. To this end, a conflict graph is maintained that gets updated for every operation to be executed. If an operation leads to a cycle in the conflict graph, this cycle is removed by aborting one of the transactions involved in the cycle. The advantage of SGT is that it permits the full concurrency of the CSR class. [WV02, p. 168]

However, the implementation of SGT for distributed systems limits the scalability of the system. All servers that process transaction operations share the same conflict graph, which necessitates complex coordination protocols. As with the other algorithms so far, SGT sends every operation to the server and thus rules out caching and highly increases latency.

Optimistic

Whereas the algorithms so far assumed that non-serializable schedules are common, optimistic algorithms are based on the opposite assumption and thus work best if only few conflicts occur [WV02, p. 170]. This assumption holds for a lot of use cases. Recall

the money transfer example where a certain amount is transferred from one account to another. Although a whole economy including multiple banks experiences a lot of transfers, each single account is updated rarely. Thus, conflicts and anomalies like lost updates are extremely rare. Nevertheless the money transfer example illustrates that even rare anomalies can have a high impact.

Optimistic algorithms apply operations without any restriction to concurrency and afterwards validate that the schedule is still serializable. According to Weikum and Vossen [WV02, p. 171], the transaction execution of the algorithm can be viewed as a three phase process:

1. *Read phase*: The transaction is executed by reading data from the database and applying writes only to a local workspace that is not visible to other transactions. The concurrency is not restricted. The result of this phase are the full read and write sets of the transaction.

2. *Validation phase*: The read and write sets are validated for conflict serializability with read and write sets of other transactions. The actual algorithms differ in how this validation is accomplished.

3. *Write phase*: If the validation was successful, the data from the local workspace is written to the database.

The validation and the write phase together must be executed as a critical section for all transactions that overlap in their accessed objects.

This separation into phases fits the requirement of this study's use case well. With the transaction applied to a local workspace in the read phase, writes do not have to be sent to the server and reads could theoretically be served from caches, which optimizes latency. For the validation and write phases, the read and write sets could be sent to the server in a single request.

The first optimistic algorithm to discuss is called **forward optimistic concurrency control (FOCC)**. In FOCC "a transaction is validated against all transactions that run in parallel, but that are still in their read phase." [WV02, p. 172]. The validation fails if one of the transactions that are currently in their read phase, has read an object that the transaction under validation is about to write. An advantage of this algorithm is that in case of a failed validation the algorithm is free to choose which transaction to abort, the one under validation or the one in the read phase.

FOCC has a number of drawbacks besides the tendency to abort long running transactions, which all optimistic protocols share. Because transactions are validated against

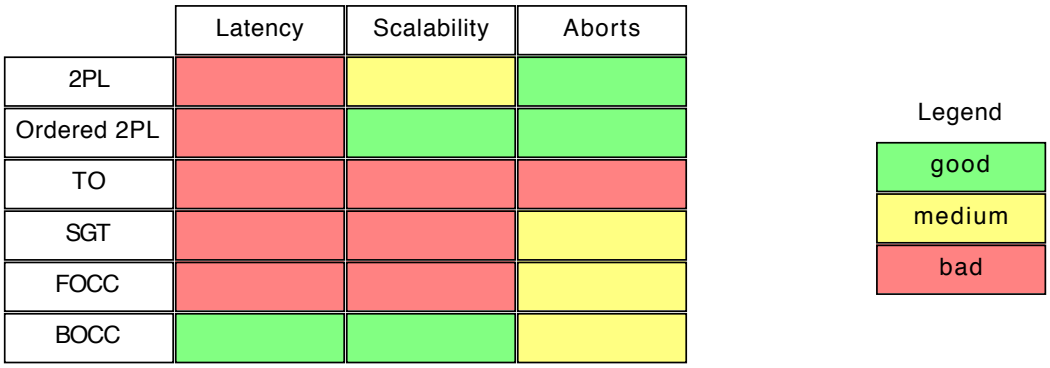


Figure 3.6: Summary of concurrency control algorithms and their properties with respect to distributed data processing

current reads, every read operation must be sent to the server, which rules out serving reads from caches and thus increases latency. Furthermore during the validation and write phases of a transaction no read operation can be executed for the objects that are validated, which severely limits concurrency and increases latency.

The second optimistic algorithm is called **backward oriented optimistic concurrency (BOCC)**. In BOCC "a transaction under validation executes a conflict test against all those transactions that are already committed" but overlap temporally [WV02, p. 172] The validation fails if an already committed transaction has written objects that the transaction under validation has read. If the validation fails, the validated transaction must be aborted. The optimization BOCC+ furthermore validates that the transaction under validation has actually missed the update of the already committed, overlapping transaction before aborting it. BOCC+ thus, eliminates aborts due to such *false conflicts* [Rah88].

In contrast to FOCC, the read phase of a transaction in BOCC can overlap with the validation and write phases of any other transaction, which improves concurrency and thus scalability to a great extend. BOCC is also well suited for caching, because operations do not have to be sent to the server and reads can be served from caches. By using BOCC, the transaction can be executed at the client with the read and write sets sent to the server at commit time for validation and to write the changes to the database.

Conclusion

The discussion in this section – summarized in Figure 3.6 – showed that optimistic concurrency control with backward oriented validation is the best fit for this study. The only problem this algorithm suffers from are aborts that especially concern long running transactions. The prevention of such aborts will be one of the main subjects of this study.

4 Distributed Cache-Aware Transactions (DCAT)

This chapter presents the transaction concept designed in this study along with important details of its implementation, a description of the recovery mechanism and an assessment of its consistency and performance properties.

4.1 The Transaction Concept

The designed transaction concept DCAT (distributed cache-aware transactions) is based on the backward oriented concurrency control mechanism, as Chapter 3 has shown that it is the best fit for the requirements identified in Chapter 2. The transaction boundaries are set dynamically by the client through *begin* and *commit* operations in the REST API. ACID semantics are enforced for all operations inside the transaction boundaries.

The transaction processing depicted in Figure 4.1 are divided into client- and server-side processing:

- **Client Side**: A transaction begin (1) is initiated by the client. Read operations (2) inside the transaction use the BFB read strategy, which serves standard requests from caches and revalidates objects according to the Bloom filter. Objects carry a version number, which is increased with every update. Write operations (3) are saved in a local workspace at the client side until commit. To commit the transaction (4), the read and write sets are sent to one of the REST servers that handles the commit procedure.

- **Server Side**: If a REST server receives a commit request containing the read and write sets of the transaction, it has to ensure the mutual exclusion (5) of the commit process for overlapping transactions. If this critical section of the commit process is entered, the read set is validated against already committed transactions. If no conflict is detected in the validation, the write set is written to the database (6) and the commit request is acknowledged (7). A recovery mechanism ensures the durability of committed transactions. If a conflict occurs, the transaction is aborted, which is signaled to the client (7).

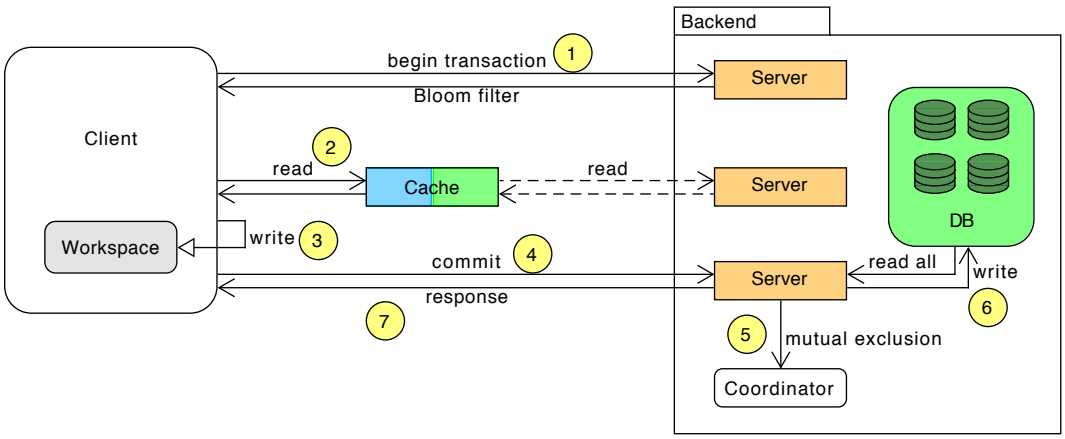


Figure 4.1: Overview of the transaction concept

The following sections describe the client and server functionality in detail.

4.2 Client Functionality

The client-side transaction processing is integrated into the client API and exposed to the user as shown in Table 4.1.

- `beginTransaction()`: This method starts a new transaction and returns a transaction context, which keeps track of the read and write sets. To start the transaction the API sends a request to the server, which returns a transaction id and updates the client's Bloom filter. The context provides all CRUD operations (create, read, update, delete) that can be executed in the transaction along with operations to commit or abort the transaction.
- `create()`: This method creates a new object that is saved in the write set inside the transaction context.
- `read()`: This method uses the read operation of the standard API with the BFB read strategy to load the referenced object. Because the Bloom filter has been refreshed at transaction begin, loaded objects are guaranteed to be at least as fresh as the transaction. The object reference and the loaded object version are saved in the read set inside the transaction context.
- `update()`: This method writes a given object by adding it to the write set.

StandardAPI Extension	
	`beginTransaction(): TransactionContext`
TransactionContext	
	`create(obj): Void`
	`read(ref): Object`
	`update(obj): Void`
	`delete(ref): Void`
	`commit(): List[Reference]`
	`abort(): Void`

Table 4.1: Client-side transaction API

- `delete()`: The delete is also handled as a write operation and the objects are added to the write set.
- `commit()`: This method sends a commit request to the server including the read and write sets. The method either returns the list of new object versions or fails with an exception, indicating a transaction abort. In either case the transaction context is closed.
- `abort()`: The abort method simply closes the transaction context and discards read and write sets. The server side does not have to be informed of the transaction abort.

To illustrate the user's workflow, consider the money transfer example from Chapter 3 in the following JavaScript listing.

```
1  var ta_context = api.beginTransaction();
2  var acc1 = ta_context.read("acc1");
3  var acc2 = ta_context.read("acc2");
4  var balance1 = acc1.get("balance");
5  var balance2 = acc2.get("balance");
6  acc1.set("balance", balance1 - 100);
7  acc2.set("balance", balance2 + 100);
8  context.update(acc1);
9  context.update(acc2);
10 context.commit();
```

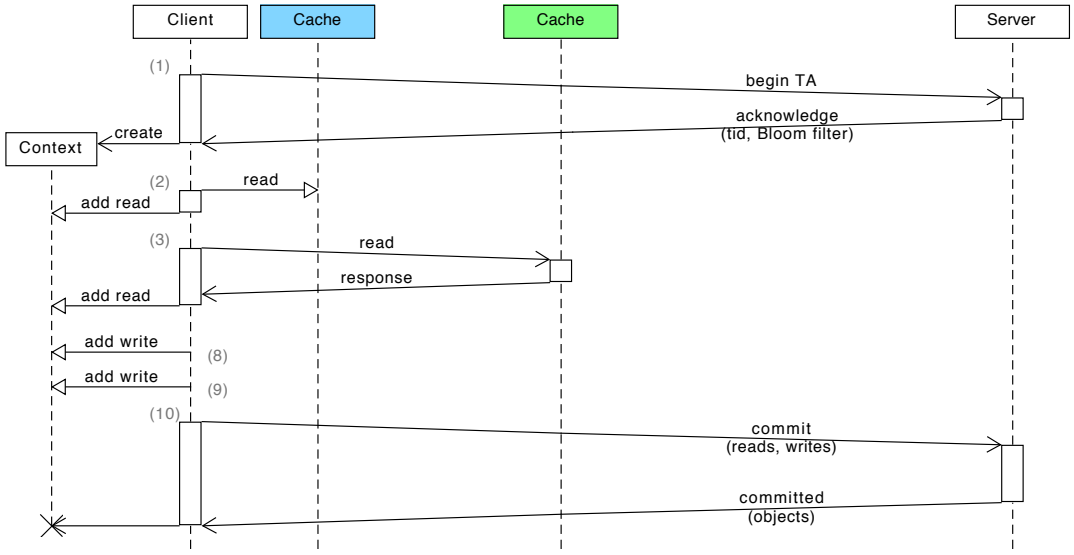


Figure 4.2: Sequence diagram of the transaction example

The sequence diagram in Figure 4.2 shows the individual steps performed by the client API. The gray numbers in the sequence diagram correspond to lines in the listing. Accordingly, Line 1 of the listing corresponds to the first client action, that connects to the server, receives the transaction id and the Bloom filter and creates the transaction context. The first account (Line 2) is returned from the client cache, the second (Line 3) from the CDN cache. Both are added to the read set. From Line 4 to 9 the balances of the accounts are altered and added to the write set to transfer $100\,\$$ from one account to the other. In Line 10 the transaction is committed, which is executed by sending the read and write sets to the server and receiving a commit confirmation along with the references of the updated objects.

The client-side transaction processing is designed to minimize latency for read and write operations while retaining a low probability of stale reads by fetching the Bloom filter at transaction begin. The server-side transaction processing is responsible for the ACID semantics, database independence, scalability as well as availability and is described in the next section.

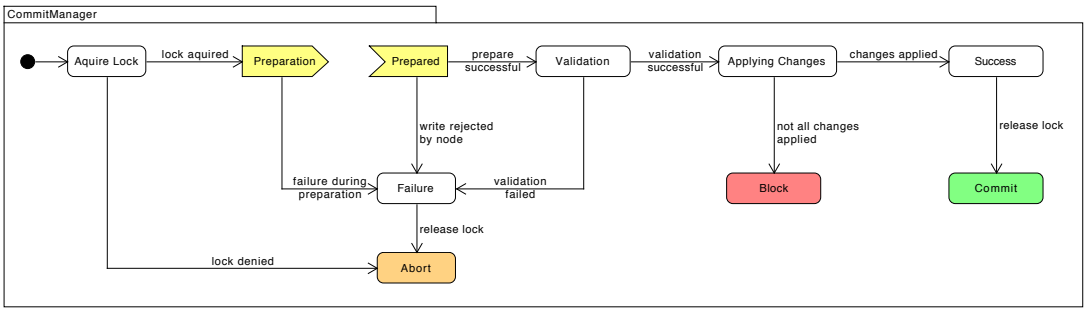


Figure 4.3: The commit process of a single transaction

4.3 Server Functionality

The REST server handles both the begin of a transaction and its commit. To start a transaction, a random transaction id is generated and sent to the client along with the Bloom filter. The much more complicated transaction commit is described in this section.

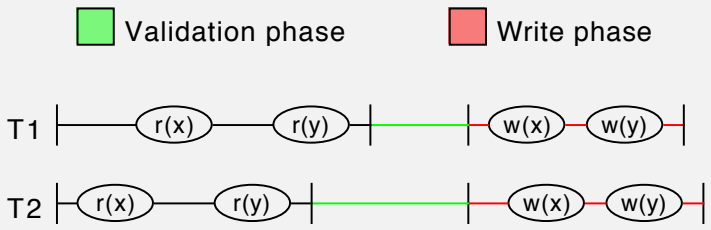
4.3.1 Commit Process

Transaction commits are load-balanced over the stateless REST servers. Thus, a commit process on a REST server handles a single transaction and has to take concurrent commit processes on other servers (or even the same server) into account. The commit of a transaction is handled by a commit manager as shown in the activity diagram in Figure 4.3.

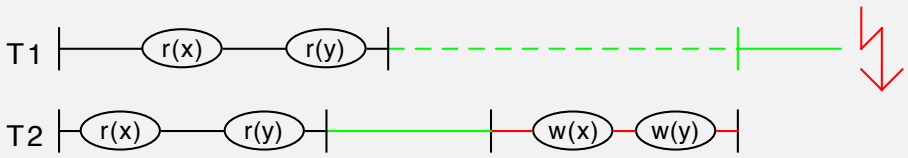
Mutual Exclusion

The first activity of the commit manager is to acquire locks to achieve the mutual exclusion of the commit processes for overlapping transactions. We define two transactions as overlapping iff they contain conflicting operations.

The Reason for the Mutual Exclusion



To understand the necessity of this mutual exclusion consider the money transfer example illustrated in the above diagram. Both transactions T_1 and T_2 try to transfer $100\,\$$ from account x to account y. Therefore both transactions read the accounts and save changes to their local write sets. At commit each transaction sends the read and write sets to the server. On the server side the read and write sets are validated and afterwards the changes are applied to the database without mutual exclusion. Because of the overlapping validation and write phase both transactions are committed successfully even though the schedule is not serializable.
The mutual exclusion of overlapping transactions is illustrated in the diagram below using the same example. In this schedule the commit process of transaction T_1 is delayed until the successful commit of transaction T_2. This way the validation can detect the conflict and abort transaction T_1, resulting in a serializable schedule.



To maximize concurrency and thus throughput of transactions on the server side, a commit process is only then delayed if the concurrent execution with another currently running process could violate serializability. As the definition of CSR (see Subsection 3.3.2) is based on conflicts between transactions, it is sufficient to prevent conflicting transactions from concurrently executing the commit process. To this end, the implementation of the mutual exclusion acquires locks for each transaction before beginning the actual commit process. For each element in the read set of the transaction a read lock and for each element in the write set a write lock is acquired. If not all locks can be acquired the

transaction's commit process is delayed and already acquired locks are released. Locks are acquired in a deterministic order to prevent deadlocks ([SGGS12]). To delay a transaction until all locks can be acquired, a retry mechanism with exponential back-off is used. If the maximum number of retries is reached, the transaction is aborted.

Because transaction commits are distributed over all REST servers, local locks are not sufficient. Instead either a distributed or a centralized service is needed. Zookeeper [Zoo] and Redlock [Reda] are both services that provide distributed locks. The current implementation, however, uses a centralized service that is simply a single Redis instance with *AOF* (append only file) persistence and *fsync* at every operation [Redb] (the Redis coordinator). This implementation ensures durable persistence of locks and has a lot less overhead compared to a distributed service. The disadvantages of the implementation are that the Redis instance is a single point of failure and a potential bottleneck for scalability. Chapter 7 contains a more detailed discussion on scalability and availability.

After acquiring the needed locks, the actual commit process begins with the validation of the read and write sets.

Validation

The validation of the read and write sets consists of multiple steps. The validation has to ensure that i) the user has the right to perform updates on the accessed tables, ii) the business logic executed in the backend does not prohibit any of the writes, iii) the user has the right to update the actual objects in the write set and iv) the execution of the transaction does not violate serializability.

The first two steps are ensured inside the so-called *stages* of the REST server. The stages asynchronously apply before-handler and register after-handler for all the writes. The before-handlers include logic to check the so-called *schema-level ACLs* and execute business logic registered at the backend, which may change the object to write or abort the write all together. The asynchronous stage logic is initiated by the **preparation** activity of the commit process (see Figure 4.3). The **prepared** activity acts on the result of this preparation. If the preparation fails – due to missing user rights for example – or a write is rejected by the client code, the transaction is aborted. Otherwise the commit process proceeds with the **validation** activity.

The **validation** activity ensures the third and fourth step. First, the so-called *object-level ACLs* are checked to ensure that the user has the right to perform the writes. If a write is rejected, the validation fails and the transaction is aborted. Second, the validation ensures that the execution of the transaction does not violate serializability. Therefore the transaction under validation must be compared to already committed transactions.

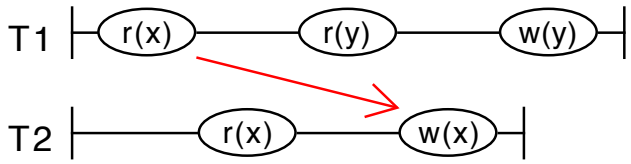


Figure 4.4: Example of an unnecessary abort

The implementation of the serializability test is quite simple [Rah88]: The versions of all read and all written objects are compared to the current database state. To ensure that the newest database state is used for the comparison, the database system has to provide a linearizable read operation [VV15]. If any of the versions differ from the current state, the validation fails und the transaction is aborted. Otherwise the validation is successful.

This validation favors simplicity over precision as it aborts some transactions even though the schedule is in CSR. Figure 4.4 shows an example of such an unnecessary abort. The depicted schedule is in CSR and equivalent to the serial schedule where T_1 is executed before T_2. Nevertheless, the validation would abort transaction T_1 because the version of object x has changed since it was read (due to the write of transaction T_2).

Even though CSR is the most commonly used correctness criteria for transactions, COCSR appears to be a better fit for cache-aware transactions as it additionally guarantees the freshness of the read objects. Because the staleness of reads is bounded by either the age of the Bloom filter (using the BFB read strategy) or the TTL of the object (using the RA read strategy), a transaction could return substantially stale data if its schedule is validated for CSR. The described concurrency control mechanism precisely matches the serializability class COCSR and is much easier than to realize classical CSR validations. Correctness and completeness is described in Subsection 4.4.3.

After a successful validation the commit process proceeds with the **apply changes** activity.

Applying Changes

The changes of a transaction are simply applied by writing the write set to the database. All objects are written with a fixed version number that is bigger than the version number of each accessed object. This way version numbers increase monotonically. Details on the version generation are described in Subsection 6.2.2.

To meet the atomicity property it is essential that either all writes are persisted or none. Thus, if the connection to the database is lost after writing some but not all objects, the commit manager has no chance but to block the commit processes of overlapping transactions by not releasing the locks. If all changes are applied, the transaction is considered as committed. Locks are only allowed to be released if the transaction is aborted (i.e. non of the writes is applied) or committed (i.e. all of the writes are applied). After releasing the locks the abort or commit is signaled to the client as the last step of the commit process.

Conclusion

The commit process consists of the main components for guaranteeing the mutual exclusion of overlapping transactions, the validation of access rights and serializability and the application on database level. The discussed implementation handles the concurrent execution of transactions and prevents intermediate states and inconsistencies.

Figure 4.5 summarizes the system parts described for the client and server side. The processing starts at the client side with the *Client API* and the *Transaction Context* and proceeds on the server side with the local *Commit Protocol* that implements the optimistic concurrency control and the *Coordinator* that ensures the mutual exclusion of overlapping transactions. The next subsection discusses an extension of the commit protocol that handles failures in the commit process and increases fault tolerance.

4.3.2 Recovery

Transaction recovery is a mechanism to ensure atomicity and durability of transactions [WV02, p. 379, 427] and also to support fault tolerance. Classical transaction processing uses *undo recovery* to *rollback* aborted transactions and *redo recovery* using transaction logs with forward rolling recovery to restore the most recent consistent database state if system components fail [WV02, p. 379f, 427ff]. The “most recent, consistent state is defined as including all updates of committed [...] and no update of uncommitted or previously aborted [...] transactions” [WV02, p. 427].

The transaction concept of this study does not need undo recovery to abort transactions because writes are not applied gradually like in classical transaction processing but rather all at once after successful validation. Thus, there is no need to undo a write operation and dirty read anomalies cannot occur either. Redo recovery on the other hand is needed to ensure the durability of committed transactions if system components fail. To this end, a *Roll-Forward Recovery* ([Mica]) mechanism based on logging and redo of transactions has been implemented and the failure scenarios in the transaction processing have been analyzed. Both are summarized in Figure 4.6.

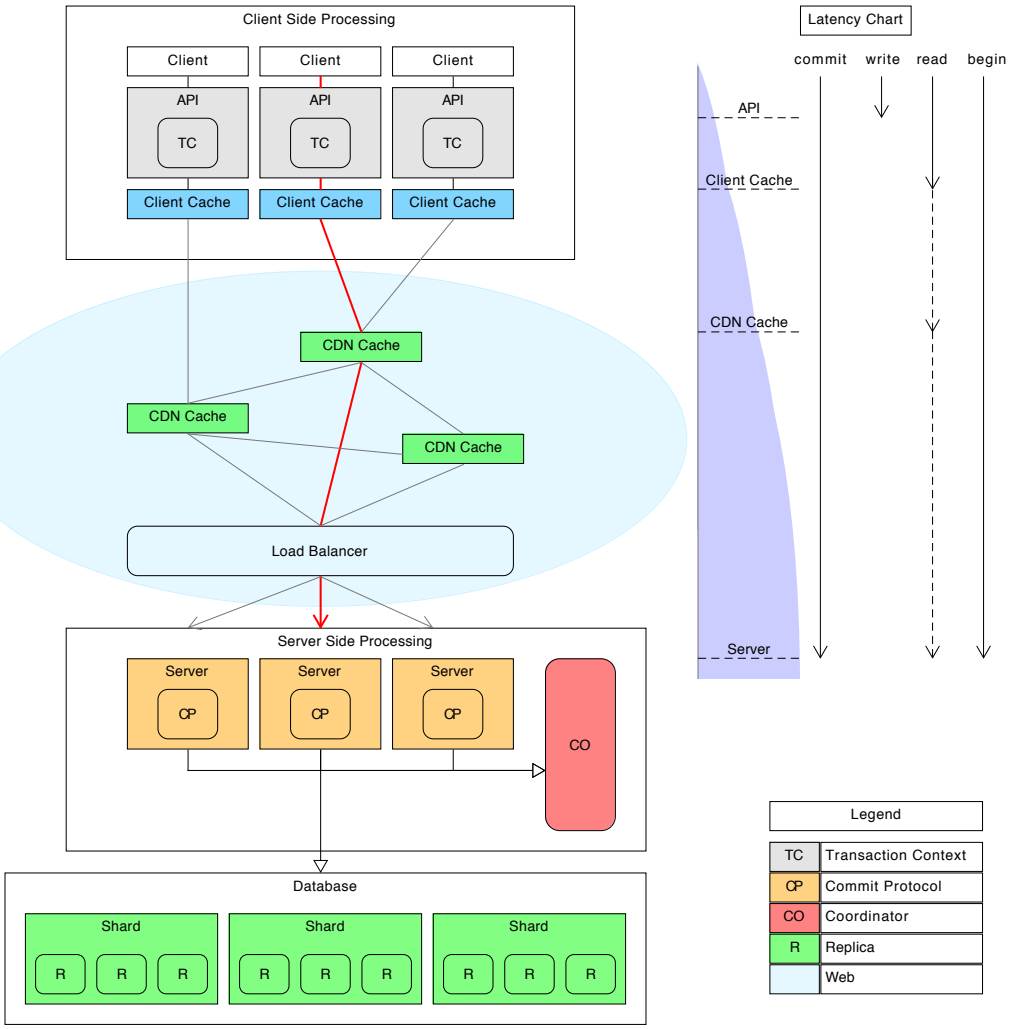


Figure 4.5: System overview of the transaction processing

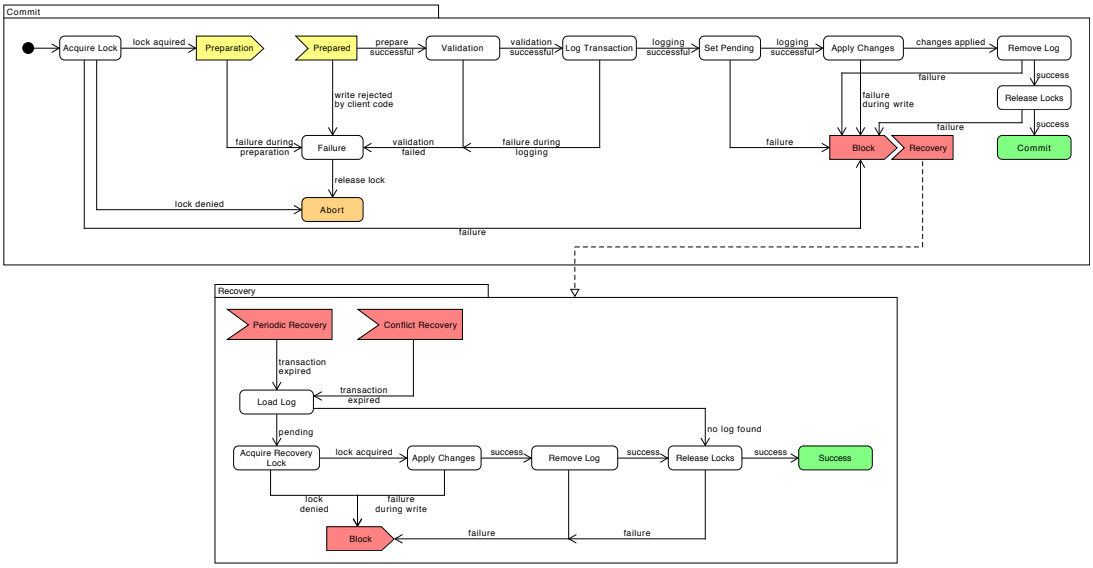


Figure 4.6: The extended commit process of a single transaction with recovery

Failure Scenarios

The following list summarizes the possible failure scenarios that can occur when the client or the server fail as a whole during transaction processing along with the implications for this processing.

1. If the client aborts the transaction or fails before the commit, the transaction will never reach the server and thus behave as if aborted. If the client fails after sending the commit the transaction processing is not affected.

2. A server failure before the **Log Transaction** activity was successful will result in the abort of the transaction. After the failure, the recovery has to release the locks.

3. A server failure after the **Log Transaction** activity was successful will result in the redo of the transaction by the recovery until the transaction is applied successfully and the locks are released. Thus, after the successful write of the log entry, the transaction will commit eventually. Note that the log entry is written after the successful validation, which is essential for correctness.

Failures of other components are handled by the commit process as depicted in Figure 4.6.

4. If the coordinator becomes unavailable during the **Acquire Lock** activity, the commit processing proceeds with the **Block** activity and signals the transaction abort

to the client. The recovery will eventually release any locks that may have been acquired during the first activity. The **Block** activity is actually not implemented as pausing the execution of the commit process, it rather terminates the process without releasing the locks. The asynchronous recovery implicitly handles the blocked commit processes.

5. If the component responsible for logging (currently also implemented in the coordinator) is not available in the **Log Transaction** activity, the transaction is aborted and the locks are released.

6. If the transaction cannot be added to the list of pending transactions the commit processing can either proceed with the **Block** activity or with the **Apply Changes** activity because the list of pending transactions is only used to optimize the recovery. Either way the transaction will commit eventually.

7. If the database becomes unavailable and not all changes can be applied, the commit processing proceeds with the **Block** activity. The recovery will apply the changes after the database is available again.

8. If the component responsible for logging becomes unavailable in the **Remove Log** activity, the commit processing proceeds with the **Block** activity. The recovery may apply the transaction again before eventually removing the log entry and releasing the locks.

9. If the component for locking becomes unavailable in the **Release Locks** activity, the commit processing again proceeds with the **Block** activity until the recovery eventually releases the locks.

Periodic and Conflict Recovery

The recovery for a transaction can be triggered in two ways, either by the *periodic recovery* or by the *conflict recovery*. The periodic recovery is a periodically running process that checks for every pending transaction if it is in the **Block** activity. In that case the recovery for that transaction is executed. The conflict recovery is run whenever a transaction cannot acquire all required locks. The transactions that are holding the requested locks are checked for being in the **Block** activity. The recovery is then executed for all blocked transactions. After the successful recovery the first transaction can try again to acquire the needed locks.

Both recovery triggers need to decide if a transaction is in the **Block** activity. This decision cannot easily be made because a server failure can block a commit process

unpredictably. Therefore the decision is made based on the runtime of the transaction i.e. if the transaction started the commit process more than 30 seconds ago, it is expected to be blocked.

The actual recovery process is the same for all transactions independent of the trigger. First the log entry that contains the read and write set of the transaction is loaded. If the log entry does not exist – which is the case in failure scenario 2 and 4 – the transaction was aborted or applied completely and the recovery releases the locks that where acquired by the transaction. If a log entry does exist, the transaction is considered to be pending and an expiring lock for the recovery is acquired. If the lock can be acquired, the recovery tries to apply the changes of the transaction. If successful the log entry is removed and the locks are released resulting in the commit of the transaction. If any failure occurs during this processing the recovery terminates in the **Block** activity. The next recovery cycle tries this process again until one eventually succeeds.

Conclusion

Blocked transactions are either recovered by the periodic recovery if they are marked as pending or if another transaction needs one of their locks (e.g. if a user checks the new state by loading the objects). The locks of failed transactions are either released before signaling the abort or in the case of a failure by the recovery as soon as another transaction needs one of the locks.

The conflict recovery may not recover all transactions that are not marked as pending – if there is never a lock conflict[1]. It ensures, however, that as long as all system components are running, blocked transactions will not prevent the execution of new ones.

4.4 Revisiting the ACID Properties

The first requirement for the transaction concept identified in Chapter 2 is called *Strong Semantics*. To show the fulfillment of this requirement this section attempts to reason about the following proposition.

> The described commit processing and recovery mechanism ensure the ACID properties for each transaction.

In the following the properties atomicity, consistency, isolation and durability will be approached separately.

[1] This behavior is similar to *ReadRepair* used in multi-master databases like Cassandra [Kun].

4.4.1 Atomicity

Atomicity means that a transaction is either applied completely or not at all and no intermediate states are visible to other transactions.

There are two cases to distinguish for the transactions' write set. In the first case, the commit processing does not write a complete log entry. This may be due to a failed validation or a failure in the logging system. In this case the writes are applied not at all. In the second case, the commit processing has written a complete log entry. In this case either the standard commit processing or the recovery mechanism will eventually apply the whole write set. Until one of the processes has written the whole write set, all overlapping transactions cannot enter the commit process. Note that there is no undo recovery that could lead to cascading aborts and aggravate atomicity enforcement.

The durability property ensures that once the write set is applied completely, no writes are lost due to any kind of failure. The visibility of intermediate states during the application of a transaction's write set is prevented by the isolation property. Both isolation and durability are discussed in following subsections.

4.4.2 Consistency

Consistency means that transactions transfer the database from one consistent state to another.

Because the transaction concept of this study does not support automatically checked consistency constraints to be specified by the user, only logically consistent transactions have to be considered. Logically consistent transactions are defined to produce a consistent database state if executed atomically and isolated on a consistent database at transaction begin. Thus, the property consistency is ensured by isolation and atomicity.

4.4.3 Isolation

Isolation means that concurrently running transactions do not affect each other. More formally, the schedule of all executed transactions must be serializable. Assuming this is already shown for BOCC, this section will show more precisely that all schedules produced by the commit processing and recovery mechanism are in COCSR (correctness) and all schedules in COCSR can be produced by the processing (completeness).

COCSR is defined as the set of schedules for which the following holds. For each pair of committed transactions t_i, t_j $(i \neq j)$ with a conflict from t_i to t_j, the commit of t_i must

have happened before the commit of t_j [WV02, p. 102f]. The argument is divided into the following two cases for correctness and completeness[2].

Correctness ($\subseteq$ COCSR)

Every schedule produced by the transaction processing is in COCSR. For the contradiction argument assume that a produced schedule contains two transactions t_i and t_j ($i \neq j$) with a conflict from t_i to t_j and that t_j has been committed before t_i.

Ignoring the recovery mechanism for a moment, it can be assumed that the mutual exclusion of the commit processes for overlapping transactions works correctly[3]. The conflict from t_i to t_j means that they did not execute the commit process concurrently. Instead t_j must have executed the commit process before t_i because we have assumed t_j committed first.

If t_j executed the commit process before t_i, there is no way t_j has read an object written by t_i. Thus, the conflict from t_i to t_j must be due to an object x read by t_i before written by t_j. That is the point where the contradiction becomes visible: If t_i reads object x in version k before t_j updates it (during the commit processing) to version n with $n > k$, the validation of t_i will abort the transaction assuming a linearizable database system. Thus, the assumption that both t_i and t_j have been committed in the schedule is wrong.

With the recovery mechanism the argument gets more difficult. The reason for the strict mutual exclusion of the commit processes of t_i and t_j without recovery is that the acquired locks do not expire and are only released in the case of a complete execution of the commit process. In the case of a failure, the locks would be held forever. That is why many locks in distributed systems expire after a timespan longer than any healthy process should hold the lock. However, if this timespan is not long enough the lock can effectively be held by multiple processes violating the mutual exclusion. A recent discussion on this problem can be found in [Kle] and [San].

Even with the recovery mechanism locks for the mutual exclusion of the commit processes do not expire and are only released once the transaction is aborted or all writes are applied successfully. A problem that, however, arises with multiple processes executing the transaction. The recovery runs periodically and can also be triggered to resolve lock conflicts. The mechanism only runs a recovery if the transaction is in the commit process for a long time and assumed to be blocked. To prevent multiple recoveries to be

[2] Completeness means the full range of schedules COCSR is accepted by the processing, which is favorable for concurrency

[3] The processes block without releasing the locks in case of a failure

performed concurrently for the same transaction, an expiring lock is acquired. Neither the prediction of a blocked transaction nor the expiring lock are safe. Thus, in rare cases there may be multiple processes working on the same transaction, i.e. applying it's write set, removing the log entry and finally releasing the locks.

Because writes are idempotent, the recovery could simply apply writes that have been applied before (by the commit process or another recovery process), but only until the locks are released. If multiple processes are working on the same transaction and the first process releases the locks after successfully applying all writes, the second process may still apply some of the writes. Thus, the second process effectively writes objects without holding a lock, hence without isolation.

The solution to this is to ensure that writes are only applied once. To this end, objects are written only if the current version in the database is smaller than the one to write (which is determined by the transaction id, see Subsection 6.2.2). The monotonically increasing object versions, the recovery and the mutual exclusion of overlapping transactions therefore ensure that each write is applied exactly once. As a consequence, releasing the locks once all writes are successfully applied by the commit or recovery process achieves correct isolation.

Completeness ($\supseteq$ COCSR)

Every schedule that is in COCSR can be produced by the transaction processing. It is to show that every transaction aborted due to the described validation process would lead to a schedule that is not in COCSR if it was not aborted.

A transaction t_i is aborted iff it has accessed an object x in version k that has afterwards been updated to version $n > k$[4]. The overlapping transaction t_j that updated x must be committed. Otherwise t_i could not have entered the commit process. Thus, there is a conflict from t_i to t_j (regarding object x) and t_j has been committed before t_i. The definition of COCSR states that there is no such constellation where t_i is committed. Thus, the commit of transaction t_i would lead to a schedule that is not in COCSR.

4.4.4 Durability

Durability means that data written by a successfully committed transaction is persisted and guaranteed to survive subsequent software or hardware failures.

The durability of the successfully applied write set is ensured on database level. The maximum durability in MongoDB is achieved with synchronous replication to ensure

[4]Aborts due to failing components or high contention on the coordinator are not considered here.

that a majority of replicas has the current database state and can replace a failing master node without losing data.

4.4.5 Implementation

A valid transaction concept that ensures ACID semantics can still be spoiled by a faulty implementation. To validate the implementation a closed economy simulation was used, similar to the one in YCSB+T [DFNR14], to calculate a transaction anomaly score for a database system.

The closed economy consists of multiple accounts with an initial balance. The invariant is that no money enters or leaves the system. Thus, the overall balance stays the same over the simulation. During the simulation different transactions are executed consisting of money transfers from one account to another, the deletion of accounts and the insertion of new ones with the same balance and simply loading of accounts.

All the transactions performed during the simulation are logically consistent with respect to the invariant. Thus, inconsistencies can only result from violating the ACID semantics. The *anomaly score* γ is calculated at the end of the simulation.

$$\gamma = \frac{|S_{initial} - S_{final}|}{n}$$

$S_{initial}$ – initial sum of all account balances

S_{final} – final sum of all account balances

n – the total number of executed transactions

The anomaly score of the DCAT implementation was zero in each of the performed simulations, even when random errors are induced in the transaction processing to test the recovery mechanism. This verification of the ACID semantics is backed up by other tests that generate schedules from random transactions (consisting of read and write operations) and check their execution for serializability. These tests confirm the correct implementation of the transaction concept and the ACID semantics.

4.5 Performance Properties

This section revisits the requirements proposed in Section 2.2 by assessing the performance properties of the DCAT transaction concept. Figure 4.7 summarizes the results of the assessment.

4.5.1 Latency

Low latency is one of the fundamental properties of Orestes and thus an important requirement for the transaction concept. The latency of a transaction, i.e. its runtime, is composed of the transaction begin, arbitrary read and write operations and the transaction commit. The focus lies on reducing the necessary connections between client and server as this has the biggest latency impact.

Transaction Begin

At transaction begin the client essentially connects to the server to fetch the Bloom filter before executing read operations[5]. Omitting this first connection to the server would be a tradeoff in favor of latency at the cost of a potentially higher risk of stale reads from caches.

Especially for short transactions with low conflict rates, omitting the first connection could be an optimization. Long-running transactions with higher conflict rates on the other hand benefit from the fresh Bloom filter, which ensures that read objects are at least as fresh as the age of the transaction.

Read

A transaction can potentially contain many read operations, making them most important for latency reduction. The BFB read strategy (see subsection 2.2.1) meets the requirements notably well. It reduces read latency substantially by using the nearest cache possible while minimizing stale reads from expiration-based caches.

More cache hits on the one hand reduce overall latency, stale reads from caches on the other hand always lead to a transaction abort. This coherence leads to different tradeoffs regarding cache strategies.

Write

Write operations are – like reads – generally highly important for transaction latency. Because writes are only applied to a local workspace before sending them to the server at commit, DCAT reduces the latency of write operations to almost zero.

[5] The returned transaction id is not needed at the client side for a correct commit processing.

Commit

The commit operation is the last step of the transaction and always needs a connection to the server to send the read and write sets. The server needs to connect to the coordinator, the database and a logging server. The implementation uses LuaScript ([Lab]) for the coordinator and bulk writes and reads for the database to combine operations and thus reduce latency on the server side.

4.5.2 Elastic Scalability

The scalability of the transaction processing concerns the server-side processing. The components of interest are the REST servers that execute the commit protocol, the coordinator responsible for the mutual exclusion of overlapping transactions and the database system with read access for the validation and write access to apply the write set.

- The **REST servers** can easily be scaled by adding new instances. The servers handle transaction commits independently and do not communicate with each other. As an effect of the stateless REST protocol the load balancer can distribute operations (including transaction commits) freely across the servers. With the load balancer in front of the REST servers it is easy to add or remove servers for elastic scalability.

- The **coordinator** that holds durable read and write locks for each transaction is currently implemented as a single Redis instance. The coordinator could be scaled by sharding the key space over multiple redis instances. The commit processing of a transaction only needs to connect to the instances that are responsible for objects in the read and write sets. The disadvantage of this approach is that the shards must be accessed successively to avoid deadlocks. This would increase the latency of the commit process. If the shards are accessed concurrently or in a non-deterministic order a deadlock detection would be needed that could decrease scalability.

- A scalable underlying **database** system is one of the assumptions of this study. The most commonly used database for Orestes is MongoDB, which is scaled by sharding and replication. Sharding is particularly advantageous for transaction processing because only shards that handle objects contained in the read and write sets must be accessed. This property of transaction processing is called *partition independence* [BFH+14]. Because of the needed linearizable read operation replication is not as effective for read scalability as in an eventually consistent setting. The trade-off between read and write latency can be tuned using quorums with different parameters [DHJ+07].

In essence, the scalability of the transaction processing is based on the statelessness of the REST server and the partition independence of the coordinator and database system.

Only the current implementation of the coordinator as a single Redis instance is not elastically scalable. However, Redis is known for its very high throughput rates which are subject to the scalability evaluation in Section 7.2.

4.5.3 Availability and Fault Tolerance

The availability of the commit processing is assessed separately for the REST server, coordinator and database system.

- The availability of the **REST server** is achieved by redundancy as described in Section 2.3.
- The availability of a **database system** is based on the replication of data onto multiple servers. To ensure a correct commit processing, the database must be able to guarantee strong consistency. Unfortunately the CAP theorem shows that strong consistency and availability cannot be guaranteed simultaneously by a distributed system in the case of a network partition [GL02]. The same argument holds for the **coordinator** that could be replicated but needs to ensure strong consistency, too.

As a consequence, there is no way to implement transaction processing with ACID guarantees in an always available fashion.

The partition independence property that enables scalability, however, also helps with availability. If one of the database partitions is unavailable, transactions that access objects from other partitions can still be processed. The same holds for the potentially sharded coordinator. The current single instance implementation of the coordinator is certainly a single point of failure that could render the whole commit processing unavailable.

Fault tolerance is another important property related to availability. The measures for fault tolerance of the transaction processing are described as the transaction recovery in Subsection 4.3.2. Most importantly the recovery ensures that as long as all relevant components are running or do recover, transactions are applied eventually and locks are released to not block other transactions.

4.5.4 Database Independence

Database independence is important for compatibility with the polyglot persistence feature of Orestes. To this end, the commit processing is implemented as a module of the

REST server using a minimal interface to the database layer. It only requires durable write and linearizable read access to the database. As a consequence, every database capable of these operations can be extended to support transaction using DCAT and Orestes.

4.5.5 Applicability

The applicability of the transaction concept to different use cases depends on its features. On the plus side, the transaction concept has full ACID guarantees for the commonly used CRUD operations inside dynamic boundaries that are set by the client through begin and commit operations. It is also scalable, recovers automatically from temporary component failures and minimizes transaction latency.

On the downside, the transaction concept does not support the execution of queries inside transactions for the lack of a protection against the phantom problem (discussed in Subsection 4.7.2), has no automatically checked user defined consistency constraints and it produces more transaction aborts due to conflicts during validation of long-running transactions compared to classical locking approaches.

The support of queries, the phantom problem and consistency constraints will be described in Section 4.7. As the problem of high conflict rates is inherent to the used concurrency control mechanism, measures against this coherence will be subject to the remainder of this study. The following section assesses the quality of this problem for DCAT.

4.6 Conflict Rates of Cache-Aware Transactions

The biggest downside of optimistic concurrency control without caching is that long-running transactions with many accessed objects have a high abort probability due to validation at commit time. The main factors on the abort probability of a given transaction are

1. the **update rate** of the read objects. A high update rate increases the probability of a conflict.

2. the **read set size** of the transaction (transaction size). Each read object adds a positive probability to cause a conflict.

3. the **runtime** of the transaction from begin to commit. The more time elapses between each read and the commit, the more likely a conflict occurs.

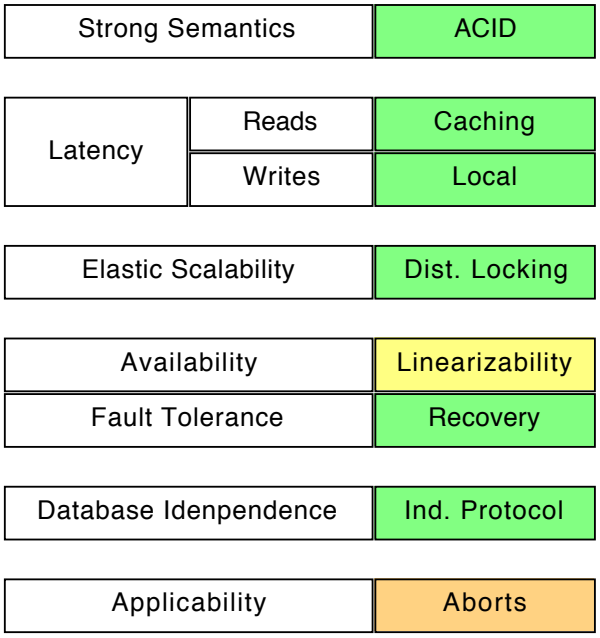


Figure 4.7: Requirements on the transaction concept and how they are fulfilled (green) or prohibited (yellow and orange)

This section describes the influence of the transaction size and runtime on the abort rate and the retried runtime and propose the key hypothesis of this study, that caching significantly alleviates the problem of high abort rates.

4.6.1 Theoretical Analysis

To gain some intuition on the expected conflict rates and their influence on the transaction runtime with retries, this subsection presents a theoretical analysis of transactions without caching based on some simplifying assumptions.

The goal is to analyze the abort probability and the runtime with retries of a read-only transaction as a function of its size. Updates are assumed to be uniformly (independently) distributed among all object and are performed at a constant rate. Time is discretized into time steps corresponding to one millisecond.

Object Update Probability

The random variable X describes whether an object w is updated in a certain time step as a Bernoulli trial. N is the number of objects in the database and r is the number of updates per time unit:

$$X(w) = \begin{cases} 1 & \text{if } w \text{ is updated} \\ 0 & \text{if } w \text{ is not updated} \end{cases}$$

$$P(X = 1) = p = \frac{r}{N} \qquad r\text{: updates per step, } N \text{ \# entities}$$

$$P(X = 0) = 1 - p = q = 1 - \frac{r}{N} = \frac{N - r}{N}$$

The corresponding binomial experiment $B_n(p)$ consisting of n stochastically independent Bernoulli trials describes whether k updates to the object happen in n time steps:

$$Y \sim B_n(p)$$

$$P(Y = k) = \binom{n}{k} P(X = 1)^k \cdot P(X = 0)^{n-k}$$

The probability that no update happens to the object in n time steps is:

$$P(Y = 0) = P(X = 0)^n = \left(\frac{N - r}{N}\right)^n$$

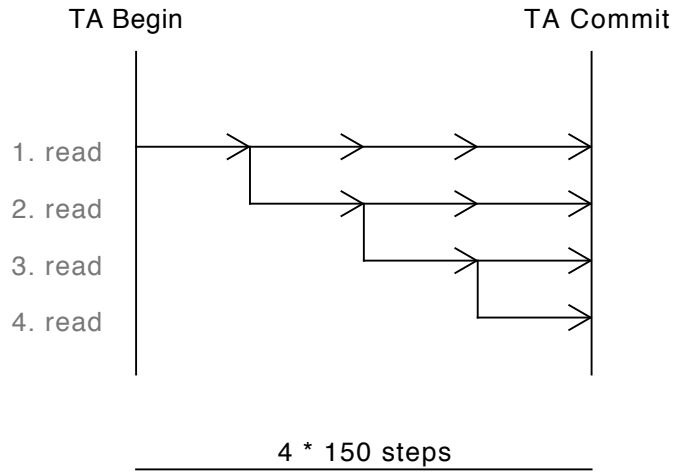


Figure 4.8: Example of the number of steps in a transaction

Transaction Composition

Another assumption is that transactional reads are uniformly distributed among all objects and performed one after the other to simulate an interactive transaction[6]. Reading an object takes m time units. A transaction aborts if any of the read objects is updated between its read and the commit of the transaction, thus the total number of steps between read and commit have to be considered for every object loaded in the transaction.

With $m = 150$ and a transaction of size four, the first read happens $4 \cdot 150$ time steps before the commit, the second $3 \cdot 150$ and so on, as shown in Figure 4.8. Accordingly, the first object has a probability to be updated between the read and the commit of $1 - P(Y = 0)^4 = 1 - P(X = 0)^{150 \cdot 4} = 1 - \left(\frac{N-r}{N}\right)^{150 \cdot 4}$.

[6] Interactive in this context does not refer to a user input, but rather an instant decision of the client program on which object to load next.

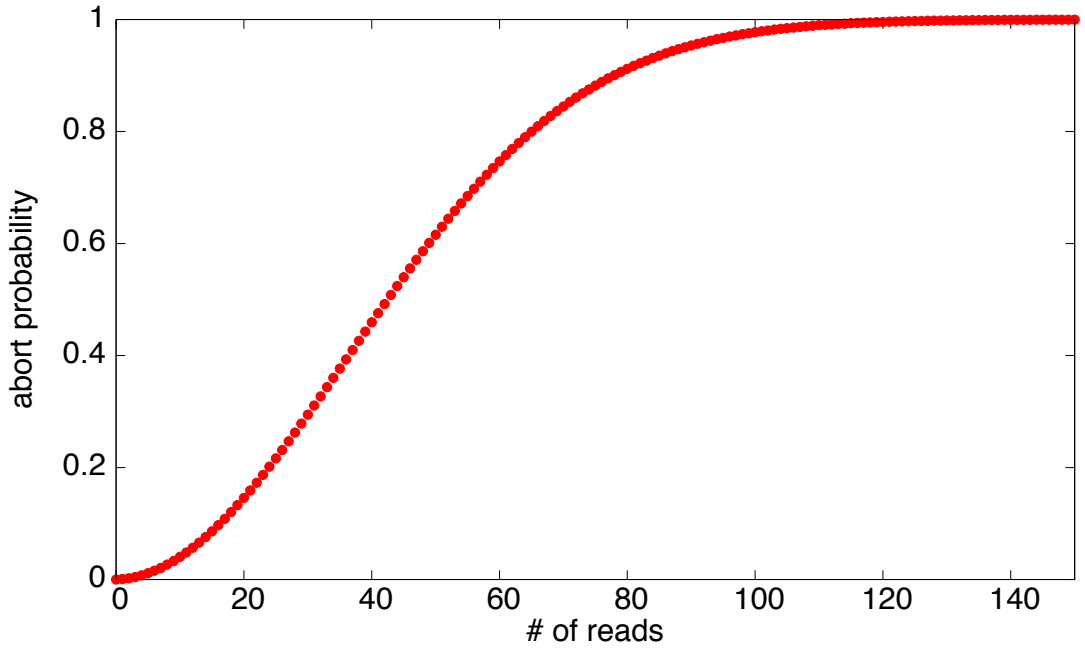


Figure 4.9: Exemplary abort rate as a function of the transaction size

Abort Probability

The random variable A describes the outcome of a transaction of size n.

$$A(w) = \begin{cases} 0 & \text{if } w \text{ is committed} \\ 1 & \text{if } w \text{ is aborted} \end{cases}$$

$$P(A=0) = \prod_{i=1}^{n} P(Y=0)^i = \prod_{i=1}^{n} 1 - P(X=0)^{i \cdot m} = \prod_{i=1}^{n} \left(\frac{N-r}{N}\right)^{i \cdot m}$$

For $N = 10000$ objects in the database, 50 updates per second and $150\,ms$ for a read operation the probability distribution is depicted in Figure 4.9 as a function of the transaction size n.

Transaction Runtime

Many use cases require a failed transaction to be retried if it is aborted due to a non-permanent cause like concurrency. The number of transaction retries R until successful commit follows a geometric distribution with the success probability $p = P(A = 0)$.

$$P(R = k) = p \cdot (1 - p)^k = P(A = 1) \cdot (1 - P(A = 1))^k$$

The expected number of retries hence is:

$$E[R] = \frac{1}{p} = \frac{1}{P(A = 0)} - 1 = \frac{1}{\prod_{i=1}^{n} \left(\frac{N-r}{N}\right)^{i \cdot m}} - 1$$

Given the time of a single transaction execution $t = m \cdot n$, the expected time of the retried execution is:

$$(E[R] + 1) \cdot m \cdot n = \frac{m \cdot n}{\prod_{i=1}^{n} \left(\frac{N-r}{N}\right)^{i \cdot m}}$$

For $N = 10000$ objects in the database, 50 updates per second and $150\,ms$ for a read operation the average runtime of a retried transaction is depicted in Figure 4.10 as a function of the transaction size n.

Conclusion

The analysis shows evidence that long-running transactions with many accessed objects on a system with a high update rate are indeed a problem.

4.6.2 Hypothesis

The key hypothesis of this study is that:

> Caching substantially improves the abort rate of transactions with optimistic concurrency control.

The hypothesis is based on the assumption that the cache integration speeds up reads substantially, resulting in a much shorter execution time. This decreases the time between reads and the commit of the transaction, resulting in a reduced abort probability. The increase in staleness by loading objects from the cache is expected to be balanced by fetching the Bloom filter at transaction begin.

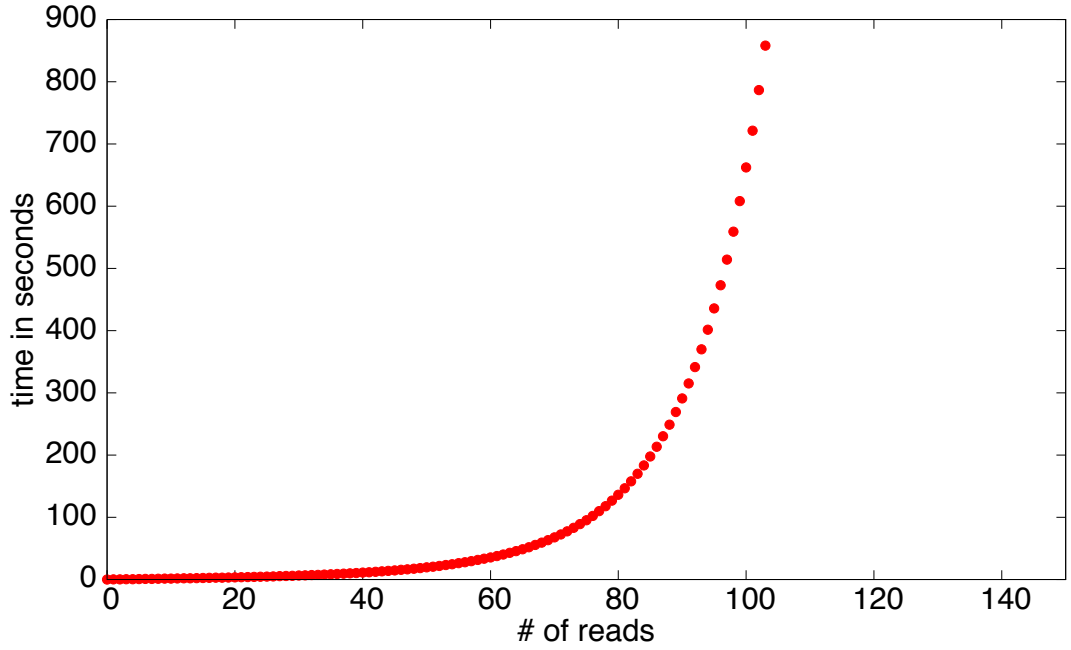


Figure 4.10: Exemplary runtime of a retried transaction as a function of its size

The hypothesis will be examined in a Monte Carlo simulation comparing transactions with and without caching in Section 7.1.

4.7 Missing Features

The DCAT implementation in its current form is missing some features necessary for certain use cases. This sections describes these features and presents ideas for a future implementation.

4.7.1 Consistency Constraints

One of the missing features are consistency constraints that can be specified by the user and are automatically checked at transaction commit. Orestes already has a user interface in its dashboard where such constraints can be specified and ensured at database level. This database level implementation does not work with the implementation of the transaction processing as persistently rejected writes are not permitted in the **Apply Changes** activity of the commit processing.

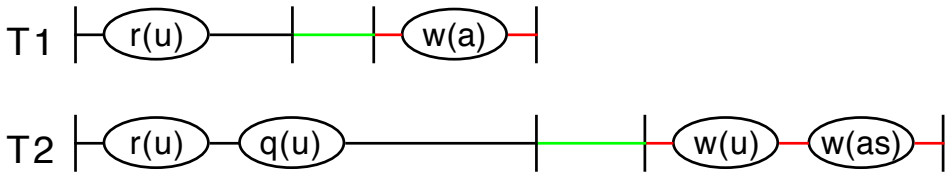


Figure 4.11: Example of a phantom anomaly

While many consistency constraints could easily be checked in the validation phase, there are some constraints that need additional coordination. Uniqueness constraints for example would need to lock the field value in question for the time of the validation.

The support for many constraints (including uniqueness) can be integrated into the DCAT implementation and is on our roadmap for future implementation.

4.7.2 Queries

Simple queries as supported by MongoDB ([Monb]) are a fundamental feature of Orestes and could easily be executed inside a transaction by adding every object from the result set to the transaction's read set. As a consequence, the commit processing would handle a query as a sequence of reads. Even though all anomalies on object level are prevented by the concurrency control, there is another anomaly that arises on query level.

To illustrate the so-called *phantom anomaly* [WV02, p. 278ff], consider the following bank account example depicted in Figure 4.11. A banking system stores users and their corresponding bank accounts in separate tables. Therefore each bank account has a field referencing the user it belongs to. If a user cancels his contract with the bank the user object and all his bank accounts are deleted. To achieve this in a transaction $T2$, all bank accounts of the loaded user must be queried ($q(u)$) and afterwards deleted ($w(as)$) along with the user object ($w(u)$) itself. A phantom anomaly occurs if another concurrent transaction $T1$ loads the user ($r(u)$), creates a new bank account ($w(a)$) and commits before the first transaction, but after the query was executed. As a result of the operations, the system contains a new account of an already deleted user.

These semantics for queries are not compatible with serializability, as no strictly serial execution of the transactions leads to this database state. If transaction $T1$ is executed

first, the newly created account would be deleted by transaction $T2$. If $T2$ is executed first, transaction $T1$ would see that the user was deleted and thus never create the new account.

Classical database systems use *predicate locking* [WV02, p. 281] to solve the phantom problem, which has a high impact on performance and scalability. Therefore the phantom problem remains an unsolved problem for DCAT in this study. Nevertheless, there are cases where one can work around the phantom problem by clever data modeling. If, in the banking example, the creation of a new account would also increase a counter (for number of accounts) in the owner's user object, transaction $T2$ would have been aborted and could be retried. The result would be a successfully deleted user with all his accounts.

5 DCAT with Partial Updates

This chapter describes the use of partial updates in combination with DCAT as an optimization to reduce abort rates of write transactions and further improve latency as well as throughput.

5.1 Partial Updates

The idea of partial updates as used in MongoDB [Mond], is to update certain parts of an object instead of writing the whole document to save bandwidth. There are simple operators like `$set` to set individual fields of objects but also more complex ones like `$inc` that can be used as read-modify-write transactions.

5.1.1 Concept

For the optimization of DCAT the complex partial update operators are of interest. They work as follows. Instead of loading an object, modifying it and writing it back to the database, complex update operators like `$inc` modify the object atomically on database level and thus prevent lost update anomalies.

For example, to deposit an amount into a bank account generally the account object is loaded, the amount is added to the current balance and the account object is written to the database. This procedure needs to be executed inside a transaction to avoid lost update anomalies. With a partial update operation like `$inc` the amount can be added on database level without loading the account object to the client. As long as the client only wants to deposit some amount and not transfer it between accounts, partial updates can be used instead of transactions.

These operators can be applied to a lot of use cases where a non-transactional read-modify-write procedure would regularly lead to lost updates. For example maintaining a counter for friends of a user or messages in an inbox (with `$inc`), adding members to a group (with `$add`) or removing an element from a queue (with `$pop`).

5.1.2 Commutativity

Many partial update operations are commutative, which means the result is independent of the order in which they are applied. The final state of a counter a for example that is incremented by two clients c_1 and c_2 is independent of the order in which the operations are applied. The same is true for a mix of increment and decrement.

$$|inc_{c_1}(inc_{c_2}(a))| = |inc_{c_2}(inc_{c_1}(a))| = |a| + 2 \qquad \text{Both increment}$$
$$|inc_{c_1}(dec_{c_2}(a))| = |dec_{c_2}(inc_{c_1}(a))| = |a| \qquad \text{Increment and decrement}$$

But not all partial update operations are commutative, i.e. the final state depends on the order of their execution. Adding and removing the same item i from a set s for example.

$$add(i, rem(i, s)) = s \cup \{i\} \neq s \setminus \{i\} = rem(i, add(i, s))$$

5.1.3 Combination with DCAT

The integration of partial update operations into DCAT is favorable for both concepts. On the one hand DCAT allows to group multiple partial update operations in a transaction to enforce ACID semantics on them. On the other hand partial update operations can substantially decrease the abort rates of write transactions in DCAT.

Complex partial update operations like $\$inc$ that automate read-modify-write transactions do not need to be validated in the commit processing of DCAT because they do not rely on a specific version of the object. The commit process of overlapping transactions is still mutually excluded though. As long as the client does not load an object to modify it and instead uses partial update operations, a validation is not needed for the involved objects. As a consequence, these operations save read requests and do not cause transactions to abort due to conflicts.

Furthermore the application of commutative partial update operations could be concurrent because the final state of the object does not depend on the operation order. As a consequence, transactions with only commutative operations could execute the commit processing completely concurrent.

In summary, the combination of partial update operations and DCAT can express complex update transactions like transferring amounts between bank accounts, without the need for validation and thus without transaction aborts. They furthermore reduce transaction latency because the account objects do not need to be loaded and also reduce the complexity of the commit processing, thus increasing the potential throughput.

Operators	Description
`$set`	Sets the object's field to the new value.
`$inc`	Increments a counter by the given value.
`$dec`	Decrements a counter by the given value.
`$put`	Replaces or sets a key value pair of a map.
`$add`	Adds an element to a set.
`$remove`	Removes an entry from a map, set or list.
`$push`	Adds an element to the end of a list.
`$pop`	Removes the last element of a list.
`$shift`	Adds an element to the beginning of a list.
`$unshift`	Removes the first element of a list.
`$replace`	Replaces the element at a given index of a list.

Table 5.1: Supported partial update operators

5.2 Implementation

5.2.1 Client Functionality

The client-side integration is quite simple. Instead of the modified object, a partial update operation is added to the transaction's write set. The update operation references the object and the field to update and contains the actual operation to perform along with all needed parameters. At transaction commit the read and write sets are send to the server which handles the commit and applies the update operations.

The set of supported update operators is similar to MongoDB ([Monc]) and includes the operators shown in Table 5.1. The commutativity of operations is visualized in Figure 5.2.

5.2.2 Server Functionality

Partial update operations are handled in the server-side commit process (Figure 4.6) similar to standard write operations as depicted in Figure 5.1.

As with standard write operations the commit process acquires exclusive locks for the objects referenced by partial update operations (even for commutative operations) and prepares the updates in the asynchronous stage logic. The validation activity, which detects conflicts by comparing read object version to the current database state and aborts conflicting transactions, is not applied to partial update operations, which reference

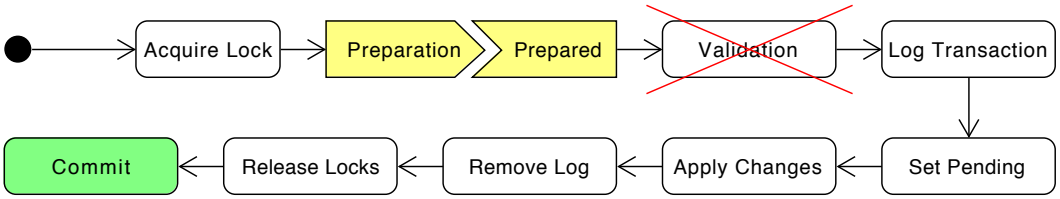


Figure 5.1: Commit process from the perspective of a partial update operation

objects without a version constraint. As a consequence, partial update operations never lead to conflicting transactions or their abort. The remaining steps of the commit process do not change for partial update operations, which are mapped to MongoDB's partial updates in the apply changes activity. For other databases the implementation of partial updates could load the objects, apply the update and write them back to the database.

Recovery

The recovery stays conceptually unchanged with the integration of partial updates. The fundamental difference, however, is that the application of transaction write sets is not idempotent anymore. An increment-operation for example must not be applied multiple times ($inc(inc(a)) \neq inc(a)$) whereas additional applications of a set-operation have no effect ($\$set(x, set(x, o)) = \$set(x, o)$).

As a consequence, it is highly important for partial updates that writes are applied exactly once. This property is already implemented in the commit process to ensure a correct recovery as mentioned in Subsection 4.4.3. Writes are applied only if the current version in the database is smaller than the one to write (which is determined by the transaction id). The monotonically increasing object versions, the recovery and the mutual exclusion of overlapping transactions therefore ensure that each write is applied exactly once.

Lock Granularity

The commutativity of partial update operations mentioned in Subsection 5.1.2 can be used to increase the concurrency of the commit processing and thus throughput and scalability. The current implementation acquires exclusive locks for objects to be partially updated. A finer lock granularity could safely allow commutative transactions to be processed concurrently.

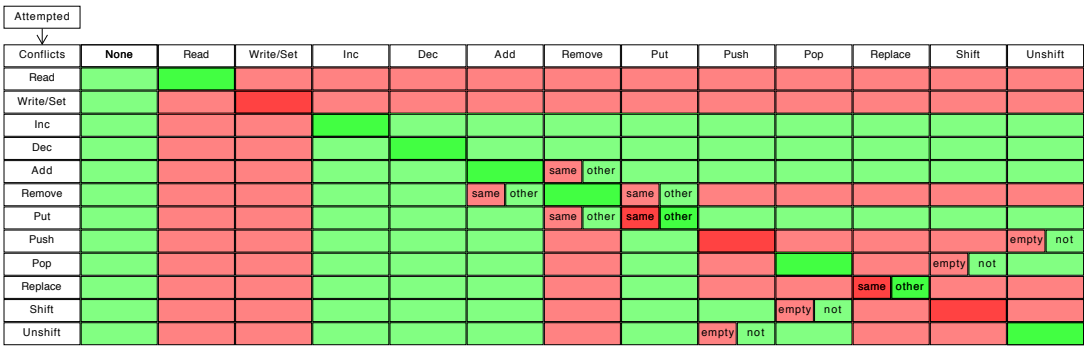


Figure 5.2: Fine grained lock mode compatibility

Commutative are i) transactions that access not intersecting sets of objects (these are already processed concurrently), ii) transactions that partially update intersecting sets of objects but not intersecting sets of object fields (currently processed serially) and iii) transaction that partially update intersecting sets of objects with intersecting sets of object fields but with commutative operations.

To achieve this fine grained concurrency control, locks of different granularity and with further lock modes can be used. Standard write and read operations would still need write and read locks on object level. Partial update operations on the other hand would only need locks on object field level. This extension of the coordinator would allow commutative transaction of type i) and ii). For transactions of type iii) lock modes on object field level must be extended to capture the actual operation type. Figure 5.2 shows the lock mode compatibilities for this extension.

Beside the complexity and increased overhead of the described coordinator extension, the concurrent execution of transactions of type ii) and iii) affects the recovery process. The guarantee that every update operation is applied exactly once is based on the assumption that version numbers increase monotonically and that an object with version n has witnessed all writes of transactions with an id smaller than n. This would no longer be true if concurrently processed transactions may update the same object. To ensure that every update operation is applied exactly once another mechanism would be needed. For example each object could maintain a list of transactions that already applied their update and only accept updates from not already included transactions.

These additional challenges have led to the decision not to implement the coordinator extension for the scope of this study and stick with exclusive locks on object level for update operations.

6 Combining DCAT and RAMP Transactions

This chapter describes the use of *Read Atomic Multi-Partition* (RAMP) transactions in combination with DCAT as an optimization to reduce abort rates of read transactions and further improve latency as well as throughput.

6.1 RAMP Transaction Concept

ACID semantics ensure database consistency for arbitrary transactions and generally come with expensive processing as well as reduced scalability and availability. In many use cases, however, certain anomalies can be accepted for the benefit of simpler processing, required by weaker isolation levels. Therefore, many isolation levels along with their respective use cases have been analyzed in research.

RAMP [BFH+14] is one of these approaches that trade classical ACID semantics for improved scalability with weaker isolation. According to the authors "Read Atomic Multi-Partition (RAMP) transactions [...] enforce atomic visibility while offering excellent scalability, guaranteed commit despite partial failures (via synchronization independence), and minimized communication between servers (via partition independence)." [BFH+14]. RAMP achieves these properties for multi-partition transactions by validating read operations at the client side and loading missing versions in a second round-trip.

A combination of RAMP and DCAT promises reduced abort rates and further improved scalability and latency for read-only transactions in use cases where atomic visibility (see next section) is sufficient.

6.1.1 Isolation Level

The isolation level of RAMP transactions is called *Read Atomic (RA)* isolation. It enforces that objects that are written together in a transaction become visible atomically. Most notably the isolation does not constrain concurrent write operations, i.e. anomalies like lost update and write skew are not prevented.

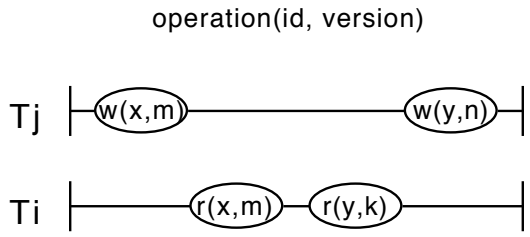


Figure 6.1: Depiction of the formal fractured read definition

This isolation level enforces for example that "if two users, Sam and Mary, become 'friends' (a bi-directional relationship), other users should never see that Sam is a friend of Mary but Mary is not a friend of Sam: either both relationships should be visible, or neither should be." [BFH+14].

Formally, "[a] system provides Read Atomic isolation (RA) if it prevents fractured reads anomalies and also prevents transactions from reading uncommitted, aborted, or intermediate data." [BFH+14]. Fractured reads are informally described in Section 3.3. Formally, "A transaction T_j exhibits *fractured reads* if transaction T_i writes version x_m and y_n (in any order with x possibly but not necessarily equal to y), T_j reads version x_m and version y_k, and $k < n$." [BFH+14] (depicted in Figure 6.1).

RAMP is not sufficient for many applications. Use cases, where RA isolation is sufficient for correctness, however, highly benefit from the increase in scalability and performance that is achieved through unrestricted write concurrency, client-side read validation and guaranteed commit. The following subsections describe how write transactions are processed and read transactions are validated.

6.1.2 Write Transactions

The purpose of RAMP write transactions is to group objects together to enable atomic reads over arbitrary shards. To this end, database partitions store multiple versions of their objects along with information which version is new, committed or stale. With each version additional metadata is stored depending on the chosen algorithm RAMP-F, RAMP-S or RAMP-H, which are described in the following subsection. To enable the atomic visibility, write operations are executed in two phases:

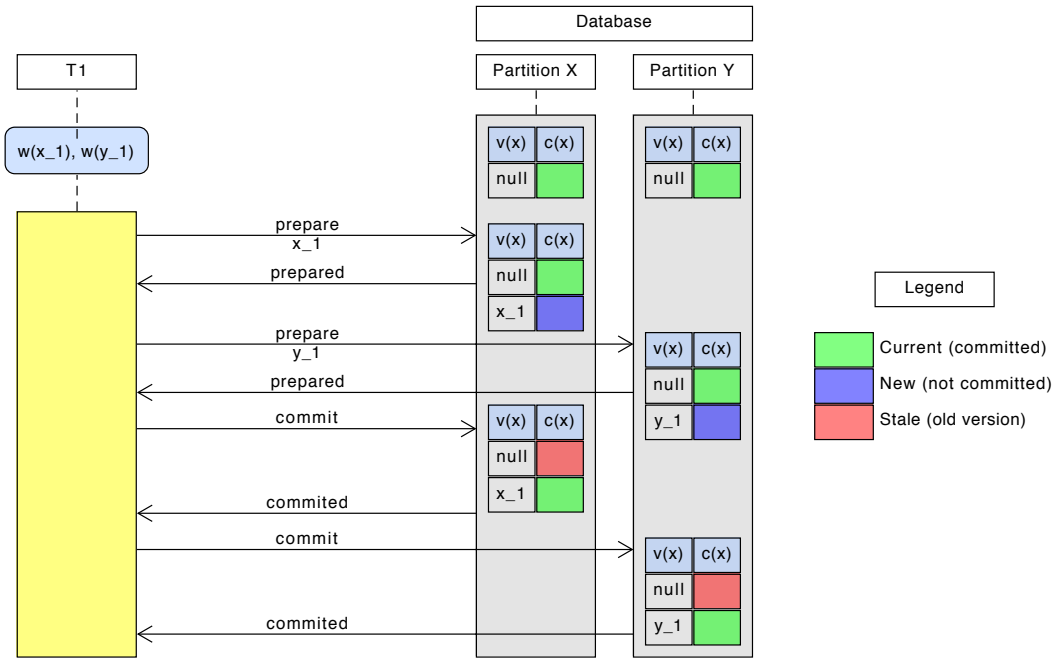


Figure 6.2: Example of a RAMP write transaction

1. **Prepare:** In the *prepare phase* the new object version is stored on the corresponding partition and marked as a new but not yet committed version. This prepare operation is acknowledged by the partition.

2. **Commit:** After all objects to write are successfully prepared, a commit message is sent to the partitions that in turn mark the prepared version as the current committed value. Instead of restricting the concurrency of writes, the transaction with the highest id supplies the committed value, which is similar to *last writer wins* as defined by Johnson and Thomas [SS05, p. 5].

The two-phase write process ensures that once a version is committed, the shards contain at least a prepared version of the objects written in the same transaction. Figure 6.2 shows an example of a RAMP transaction T_1 writing two objects x_1 and y_1 on different partitions in a sequence diagram. $v(x)$ denotes the version of the object and $c(x)$ the commit state where green means current committed value, blue means prepared but not committed and red means old value.

6.1.3 Read Transactions

The way RAMP validates read-only transactions and reloads missing versions depends on the selected algorithm.

- **RAMP-F**ast minimizes the number of round-trips needed to achieve RA isolation. It assumes that along with every written object the ids of object written by the same transaction are stored as metadata. In the first round-trip the current committed version for each object is loaded along with the metadata. Based on this metadata the client can decide whether a fractured read has happened. Referring to the definition of fractured reads, the metadata loaded with object x_m reveals that the read of object y_k has led to a fractured read, because transaction T_j, that wrote x_m has also written y_n with $n > k$. Potential fractured reads are resolved in the second round-trip by explicitly loading the missing version - in this case y_n (which is not necessarily committed). If no fractured read occurs, RAMP-F needs one round-trip, otherwise a second round-trip is needed to resolve all fractured reads. This round-trip optimization comes at the cost of per object metadata size linear to the transaction size.

- **RAMP-S**mall is the simplest algorithm and always needs two round-trips. It only assumes that along with every written object the transaction id is stored. In the first round-trip the current committed versions of the objects are loaded. In the second round-trip the list of transaction ids is used to load a newer version (not necessarily committed) of the objects written by one of the listed transactions. This way all potential fractured reads are resolved. This algorithm always needs two round-trips, even if no fractured reads occur, but has the benefit of constant metadata size per object.

- **RAMP-H**ybrid combines the advantages of the former two algorithms. Instead of storing the all ids from the write set of the transaction with each object (as in RAMP-F), it uses a Bloom filter to encode the metadata. The detection of fractured reads works like in RAMP-F but with highly compressed metadata. The algorithm usually needs one round-trip if no fractured reads occur and a second to resolve them. Rare false positives of the Bloom filter may trigger not needed second round-trips though.

Figure 6.3 extends the example from Figure 6.2 to show how a fractured read occurs and how it is resolved in RAMP-F. The figure shows a writing transaction T_1 and a reading transaction T_2. Due to the concurrent execution T_2 exhibits a fractured read by reading

the committed version x_1 from partition X while reading a $null$-value (aka. y_0) from partition Y (because y_1 is not committed). The validation detects the fractured read by comparing the loaded versions with the expected ones, based on the retrieved metadata. In this case the transaction that wrote x_1 has also written y_1, which was loaded in version 0. As a consequence, the fractured read is resolved by explicitly loading object y in version 1. Note that the two-phase write ensures that version y_1 is stored at partition Y before any object written by transaction T_1 is marked as committed.

Because RA isolation only concerns read operations, RAMP lends itself well to be used in combination with Commutative Replicated Data Types (CRDTs [SPBZ11]), which can ensure consistency for certain concurrent write operations. Though CRDTs are an effective and scalable concept to handle concurrent write operations, they are limited to certain use cases like maintaining counters or sets of items and cannot replace ACID semantics for general read-modify-write transactions.

6.1.4 Combination with DCAT

The strengths of both DCAT and RAMP complement each other especially well. RAMP benefits from the strong semantics of DCAT especially to avoid write anomalies. DCAT on the other hand benefits from RAMP's client-side validation to avoid the server-side commit processing and from RAMP's guaranteed commit to decrease abort rates for read-only transactions.

The way both concepts could work together is that,

- **DCAT** is used for all transactions that require ACID semantics and especially to isolate concurrent write operations whenever needed. The use of DCAT's partial updates similar to RAMP using CRDTs for read-modify-write transactions. To retain the foundation for RAMP's client-side validation, DCAT transactions need to store multiple versions and metadata like RAMP transactions do.

- **RAMP** is used for read-only transactions where RA isolation is sufficient. These transactions could load objects from cache just like in DCAT and validate the result set at the client side without contacting the server. Instead of aborting the transaction in the case of a fractured read (like DCAT would), the missing versions would be explicitly loaded from the server.

This combination of DCAT and RAMP would overcome RAMP's problem with lost updates, solve DCAT's high abort rates for long-running read-only transactions for which RA isolation is sufficient and further improve both transaction latency and throughput.

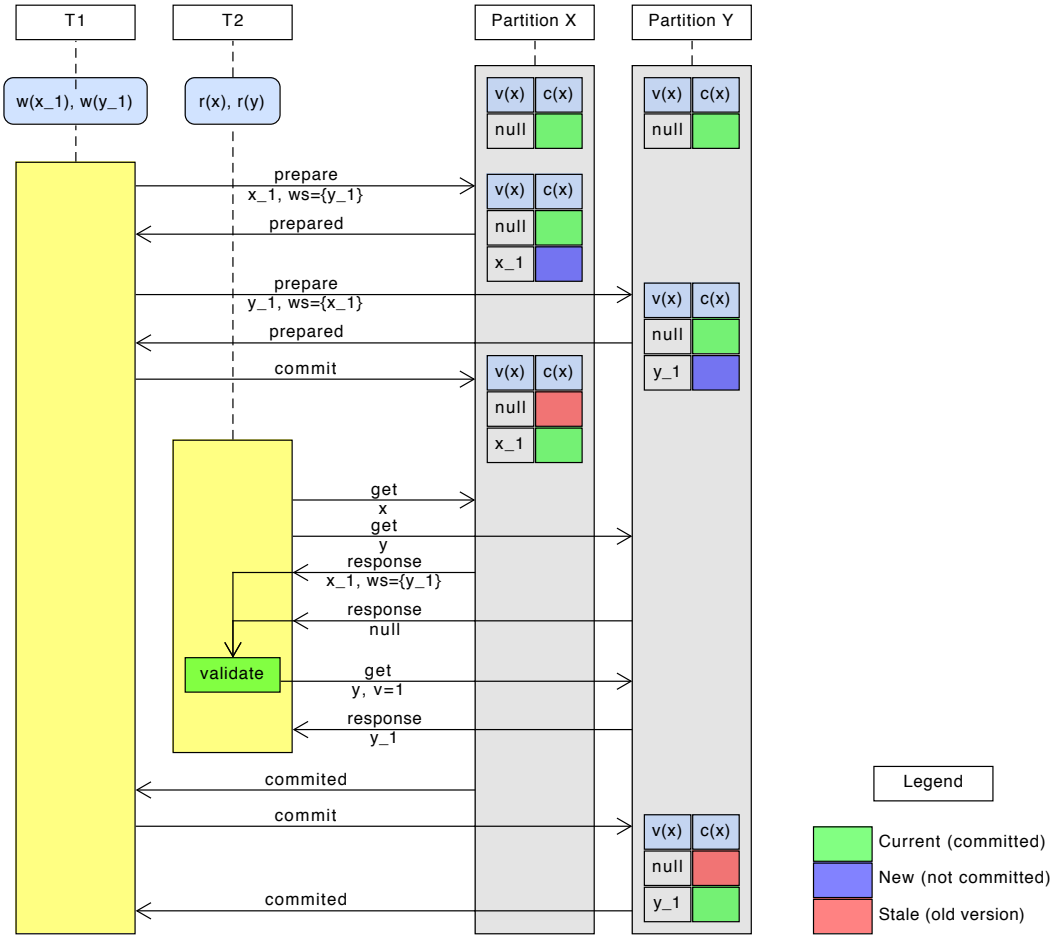


Figure 6.3: Example of a fractured read resolved by RAMP

6.2 Implementation

The RAMP algorithm integrated into DCAT is RAMP-F (can easily be extended to RAMP-H), which can be divided into two components. First, the client-side component responsible for read-only transactions served preferably from caches, validation for fractured reads and resolving them by explicitly loading missing versions from the server. Second, the server-side component responsible for storing multiple versions of an object together with the metadata needed for the client-side validation and executing the two-phase write operations. The client and server components are described in the following subsections.

6.2.1 Client Functionality

The client interface consists of an operation to load a set of objects for which RA isolation is enforced by detecting and resolving fractured reads. To this end, all referenced objects are loaded preferably from caches using the BFB read strategy (see Subsection 2.2.1). Based on this intermediate result set fractured reads are detected as shown in Listing 6.1.

```
1 def DETECT_FRACTURES(resultSet)
2   for obj in resultSet do
3     expVersion = obj.version
4     for other in resultSet do
5       if other.writeSet.contains(obj.id) then
6         expVersion = max(expVersion, other.version)
7     if expVersion > obj.version
8       yield (obj, expVersion)
```

Listing 6.1: Algorithm for fractured read detection

To detect fractured reads, the expected version for each object is compared to the actual loaded version (Line 7). If the loaded version is smaller, a fractured read has occurred and the stale object is added to the result set (`yield`) along with the expected version (Line 8). The expected version of an object `obj` is the highest version of all other objects whose metadata (`writeSet`) contains the object `obj` (from Line 4 to 6). This effectively checks if another transaction that wrote one of the objects from the intermediate result set has also written a new version of the object `obj`. This procedure is valid because all objects written by the same transaction get the same version as described in Subsection 4.3.1.

All detected fractured reads are resolved by explicitly loading the expected versions (line 7) of the stale objects in one request. This client-side implementation needs one round-trip in the worst case and zero round-trips in the best case assuming cache hits in the first read phase.

6.2.2 Server Functionality

The server-side implementation is responsible for the two-phase write processing and storing multiple object versions including the transaction metadata. The metadata in form of object ids written by the same transaction set is simply stored as a field of each object like the object's version.

Multi-Version Support and Two-Phase Writes

To store multiple versions, the RAMP write processing assumes a multi-version database system, which is not required for Orestes and thus would restrict the polyglot persistence feature. Moreover, the commonly used database for Orestes is MongoDB, which has no native multi-version support. As a solution, a database independent implementation for multi-version support could be implemented using a *look–aside-store* which stores prepared and old versions and switches them at commit. The DCAT integration, however, utilizes a more optimistic procedure for simplicity.

Instead of changing the write processing to use two phases and store multiple versions, the request for an explicit object version is answered using the current database value (if it matches the version) and utilizing the recovery logs as a fallback. The assumption of the procedure is, that in most cases where a fractured read is detected, the explicitly loaded newer version of the object is stored in the database anyway and the fallback is rarely used. Thus, the common operations like writing objects and loading the currently committed version are optimized in favor of rare operations that load a prepared or old version. To this end, the RAMP integration does not change the write procedure from Subsection 4.3.2 at all and instead processes the request for an explicit version of an object as follows:

1. The current committed version of the object is loaded from the database and returned if its version matches the request.

2. Otherwise, the transaction log (used only for recovery so far) is scanned for the transaction that wrote the requested version of the object.

3. The object is extracted from the transaction's write set and returned to the user.

Unfortunately this procedure does not work with partial updates because the partial update operations stored in the transaction log are not entire objects that could be returned. A similar problem would arise when using the look-aside-store multi-version support.

The problem can simply be solved by aborting transactions that request transaction logs containing partial update operations and retrying them[1].

Transaction Timestamp Generation

The transaction timestamp plays a fundamental role for both DCAT and RAMP and is used as a version number for all written objects.

The processing of RAMP read transactions relies on the fact that versions increase strictly monotonic for each object. In order to ensure this constraint the transaction id must be greater than any version of the objects to write by that transaction. To this end, the commit processing draws a new transaction id after acquiring the locks. The id is a monotonically increasing number returned by the coordinator that acquires the locks. The single Redis instance ensures the strict monotonicity of the transaction ids and the mutual exclusion of the commit process for overlapping transaction. This prevents race conditions of overlapping transactions that could lead to non-monotonic object versions.

Strictly monotonic object versions are not only needed for RAMP transactions but also to ensure the correct recovery (mentioned in Subsection 4.4.3) by preventing transaction writes to be applied multiple times. This is especially important in connection with partial updates. Furthermore the constraint and its implementation do not restrict the scalability of the coordinator, as a distributed (sharded) coordinator could receive a monotonically increasing number from every accessed shard and return the maximum.

Garbage Collection

Garbage collection is the first aspect where the integration of RAMP transactions makes the transaction processing of DCAT more complex. Former to the RAMP integration transaction logs were kept until the persistent execution of all write operations was ensured and deleted afterwards. With RAMP read transactions that may explicitly request old object versions, transaction logs must be kept longer. Caching even aggravates the problem because the probability that a fractured read involving stale objects occurs is increased.

The solution proposed in [BFH+14] defines a static GC window after that old versions are removed from the database. If a transaction runs longer than the GC window it may request an already deleted object version and is aborted. While the standard RAMP GC window corresponds to the maximum transaction runtime, an equivalent for the DCAT integration should correspond to the maximum TTL, because stale objects are served

[1] Another strategy to implement RAMP in single version databases is to always abort and retry the transaction if a specifically requested version is not found in the database.

from caches. The current DCAT integration, however, simply uses a static GC window for transaction logs (of committed transactions) in general and thus may lead to more transaction aborts.

6.3 Guarantees and Performance

To illustrate the consistency guarantees and performance achieved with RAMP transactions consider the following test based on the closed economy workload from YCSB+T [DFNR14].

As described in Subsection 4.4.5, the closed economy consists of a number of bank accounts, each with an initial balance. During the simulation DCAT transactions are used to transfer money between accounts. Periodically all accounts are loaded to sum up the balance, which is then compared to the initial sum of balances. The example compares the consistency and perceived performance of different strategies to load the accounts using a simple single server setup.

- **Standard read operations:** Loading the account objects without considering transaction boundaries is fast ($250\,ms$) and never aborts. On the other hand it does not return consistent results (anomaly score 0.5, see Subsection 4.4.5).
- **DCAT transaction:** Loading the account objects inside a DCAT transaction always returns a consistency result (anomaly score 0) but has a high abort probability. The retry of aborted transactions (3 tries on average) results in a highly increased execution time ($2600\,ms$).
- **RAMP transaction:** Loading the account objects in a RAMP transaction uses standard read operations in phase one and resolves fractured reads by explicitly loading missing object versions in the second phase. The RAMP transaction never aborts and its average latency ($260\,ms$) is similar to the standard read strategy. Unfortunately, RA isolation is not suited to achieve full consistency in this use case and the resolved fractured reads barely have an effect on the resulting anomaly score (0.5, same as standard read operations).

This example is supposed to show two things. First, RAMP read-only transactions have the potential to solve the abort rate problem for long-running read transactions, highly decrease latency and improve scalability. Second, RAMP can only be used in special use cases (or with special data models as will be shown in Section 6.4) where RA isolation is sufficient for correctness and not in general for read-only transactions.

6.4 Finding Use Cases

Understanding RA isolation well to find relevant use cases and correct data models is essential for the use of RAMP. This section analyzes different use cases and data models with regard to RA isolation.

Simple Data Models

The introductory example for RAMP use cases is about *Sam* and *Mary* becoming friends with the constraint that "other users should never see that Sam is a friend of Mary but Mary is not a friend of Sam" [BFH+14]. One way to model this bi-directional relationship would be a simple user object containing a list of friend ids pointing to their user objects. To establish the relationship, a DCAT transaction would add Sam's id to Mary's friend list and vise versa. A RAMP read transaction loading Mary's current user object and an old user object of Sam would identify the fractured read and reload Sam's current user object, satisfying the constraint above.

For this simple example RA isolation is sufficient but introducing an additional user *Jim* can lead to inconsistencies. Figure 6.4 shows an example where transaction T_1 establishes the friendship between Mary and Sam and T_2 another friendship between Sam and Jim. The RAMP read transaction T_3 reading Mary in version 0 and Sam and Jim in version 2 does not exhibit fractured reads (according to the definition), because it reads both objects written by T_2 and no object written by T_1 (see Figure 6.4). Nevertheless the constraint is violated – Sam references Mary as his friend, while Mary has no friends at all. The second transaction T_2 hides the fractured read that would otherwise appear between transaction T_1 and T_3, which is why this anomaly will be called *hidden fractured read* from here on.

The same problem arises in the closed economy example from Section 6.3. Figure 6.5 shows two transactions T_1 and T_2 transferring $1\,\$$ from account A to B and then from B to C. The RAMP transaction T_3 reads account A in version 0 and B and C in version 2. Again, T_3 exhibits no fractured reads but returns an overall balance that is off by $1\,\$$.

A similar anomaly arises if the RAMP transaction only reads accounts A and C. Account A and C may belong to the same company and money is transferred from account A over B to Account C. In this case loading account A in version 0 and account C in version 2 would show a higher balance than what has ever been on the companies accounts. This anomaly depicted in Figure 6.6 shows a transitive conflict cycle, which is why it will be called *transitive fractured read* from here on.

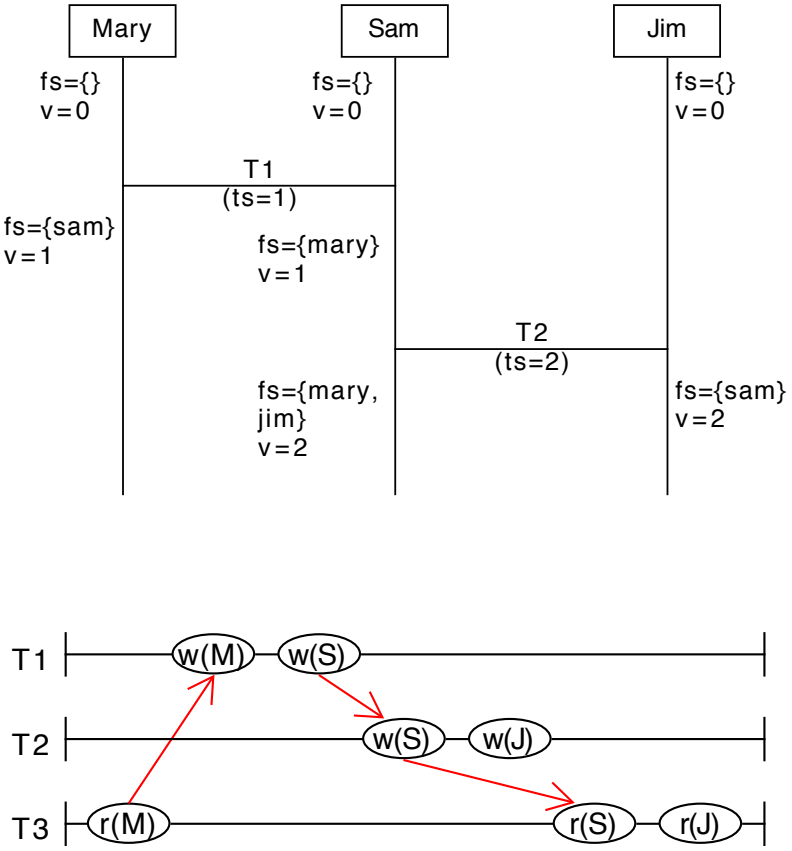


Figure 6.4: Example of a hidden fractured read anomaly in the context of bi-directional relationships

The difference between hidden and transitive fractured reads becomes clear by comparing the schedules in Figure 6.5 and Figure 6.6. A hidden fractured read is characterized by the fact that by removing one of the transactions (transaction 2 in Figure 6.5), a fractured read (according to its definition) becomes visible. Thus, the fractured read was hidden by the transaction. A transitive fractured read on the other hand contains no hidden fractured reads, i.e. removing transactions from the schedule does not expose a fractured read. This factor makes is even harder to detect transitive fractured reads.

Similar consistency problems caused by hidden fractured reads and transitive fractured reads occur in simple data models for bi-directional relationships (foreign-key-

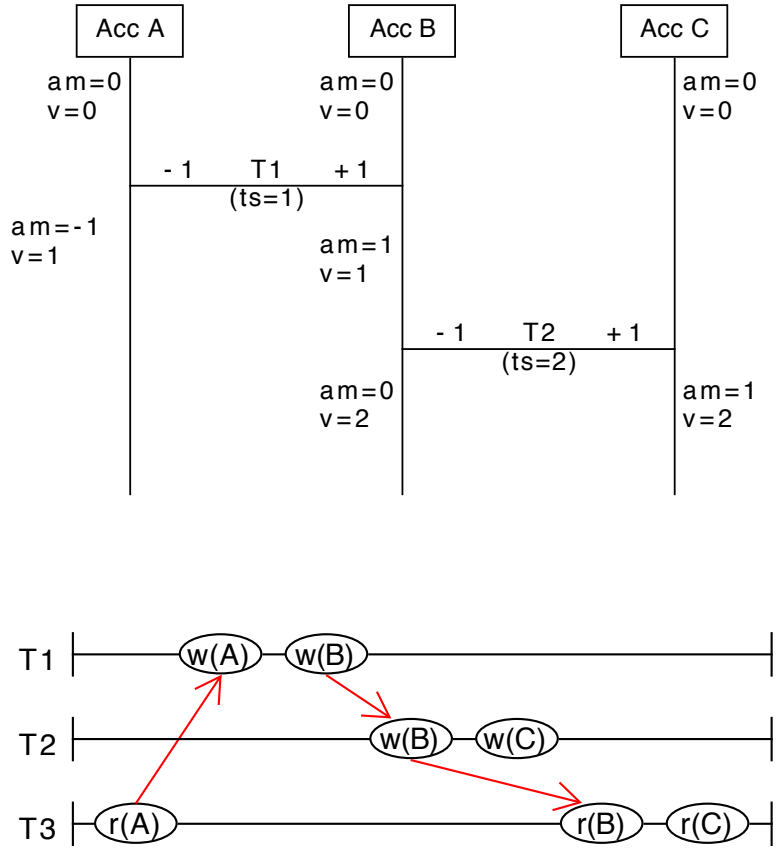


Figure 6.5: Example of a hidden fractured read anomaly in the context of materialized view maintenance

constraints) and materialized view maintenance, which are two of the three use case classes identified in [BFH+14].

Fixing the Data Model

After contacting the main author Peter Bailis, he proposed a different data model to solve the hidden fractured read problem. The friend list is not contained in a complex user object with its own version anymore. It is now rather a virtual list of friend relationships where each insertion or deletion to or from this list is a separate object with an id and

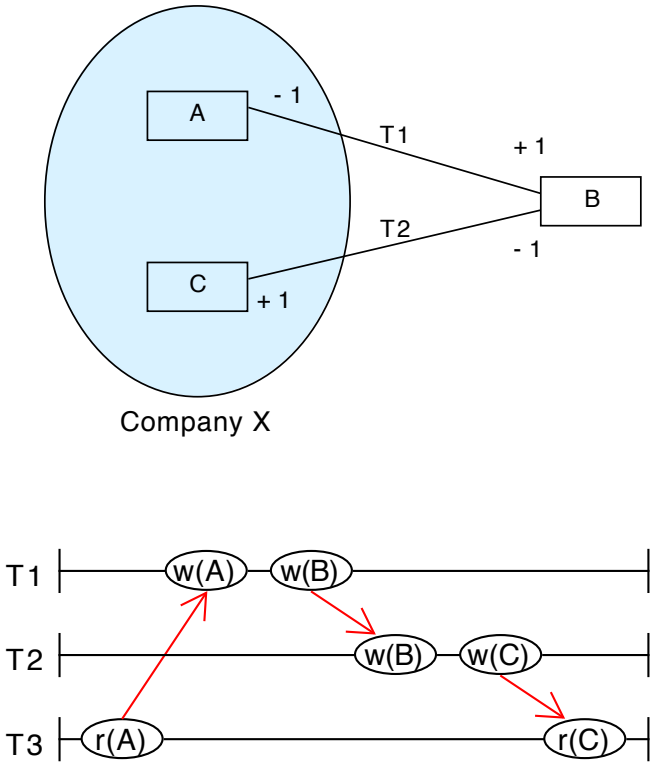


Figure 6.6: Example of a transitive fractured read anomaly in the context of materialized view maintenance

transaction metadata. CRDTs [SPBZ11] work in a similar way, as they store an entry for each change event (insertion and deletion for example) in the CRDT data-structure. Figure 6.7 shows the friend list items created by transaction T_1 and T_2 establishing the friendship of Mary and Sam or Sam and Jim respectively. Loading a friend list is achieved by querying all relationship entries containing the specified user.

To revisit the hidden fractured read problem from Figure 6.4, the equivalent scenario for this data model is the following. The query for Mary's friends returns no object, for Sam's friends both objects are returned and for Jim's friends the one existing object is returned. Executed inside a RAMP read transaction the metadata of this intermediate result set is analyzed. The metadata of object `s.m` refers to an object `m.s` written by

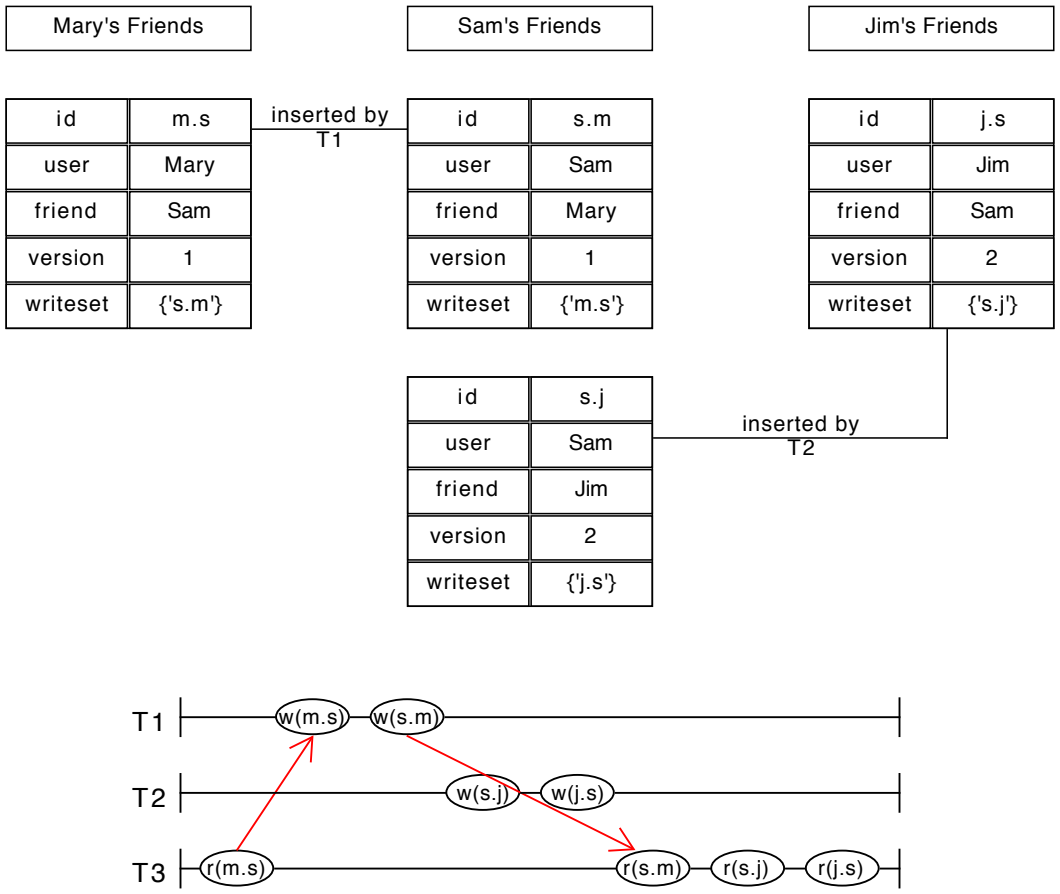


Figure 6.7: Example of data model using friend list items as objects to avoid hidden fractured reads

the transaction T_1 but not loaded by the query for Mary's friends[2]. The fractured read becomes apparent when visualizing the loaded relationship entry versions in Figure 6.7. Finding no object for Mary is equivalent to loading version 0 of the item `m.s`. The fractured read is detected because the query for Sam returns the object `s.m`. The fractured read is resolved by explicitly loading object `m.s` in version 1.

This kind of data modeling solves the hidden fractured read problem at the cost of an additional object per insert (and delete) operation plus metadata for each, compared to

[2]The owner of the friend list item is encoded in the id (`m.s`).

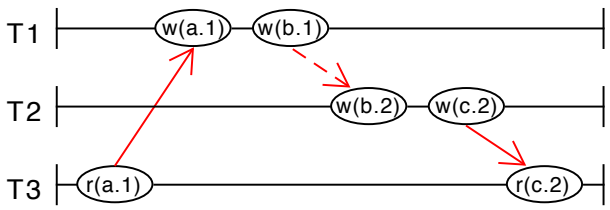
the first data model. The complexity of the data model, however, makes it hard to understand and adapt to other use cases. To solve these problems, the RAMP approach could be integrated into CRDTs, which encapsulate the data model complexity and perform advanced garbage collection to decrease the storage overhead.

The Remaining Problem

The same kind of data modeling can be applied to the closed economy use case. The balance of a bank account is not a single value inside the account object but composed of multiple transfer records instead. Every change of the amount creates a new object keeping track of the amount delta to add to the overall account balance. Figure 6.8 shows the transfer records for the two money transfers T_1 and T_2 from account A to B and B to C respectively. Reading an account balance is achieved by querying all transfer records of that account and summing up the amount values.

The hidden fractured read problem from Figure 6.5 is resolved similarly to the friend list use case. For account A no record is found, which is equivalent to loading record `a.1` in version 0. For account B and C all three records are loaded. The metadata of record `b.1` reveals the fractured read that is then resolved by loading record `a.1` in version 1.

While the advanced data model does prevent hidden fractured reads, transitive fractured reads remain a problem. Recall the example of the company reading the balance of their bank accounts A and C in Figure 6.6. The RAMP transaction that finds no record for account A but loads the record for C (shown in the following Figure) exhibits no fractured read but still returns an overall balance off by $1\,\$$.



Conclusion

The problem of transitive fractured reads (*transitive happens-before relationships*) has been discussed in [BFH+]. The only way to achieve serializability for arbitrary RAMP read transactions is to extend the protocol to track dependencies across transactions.

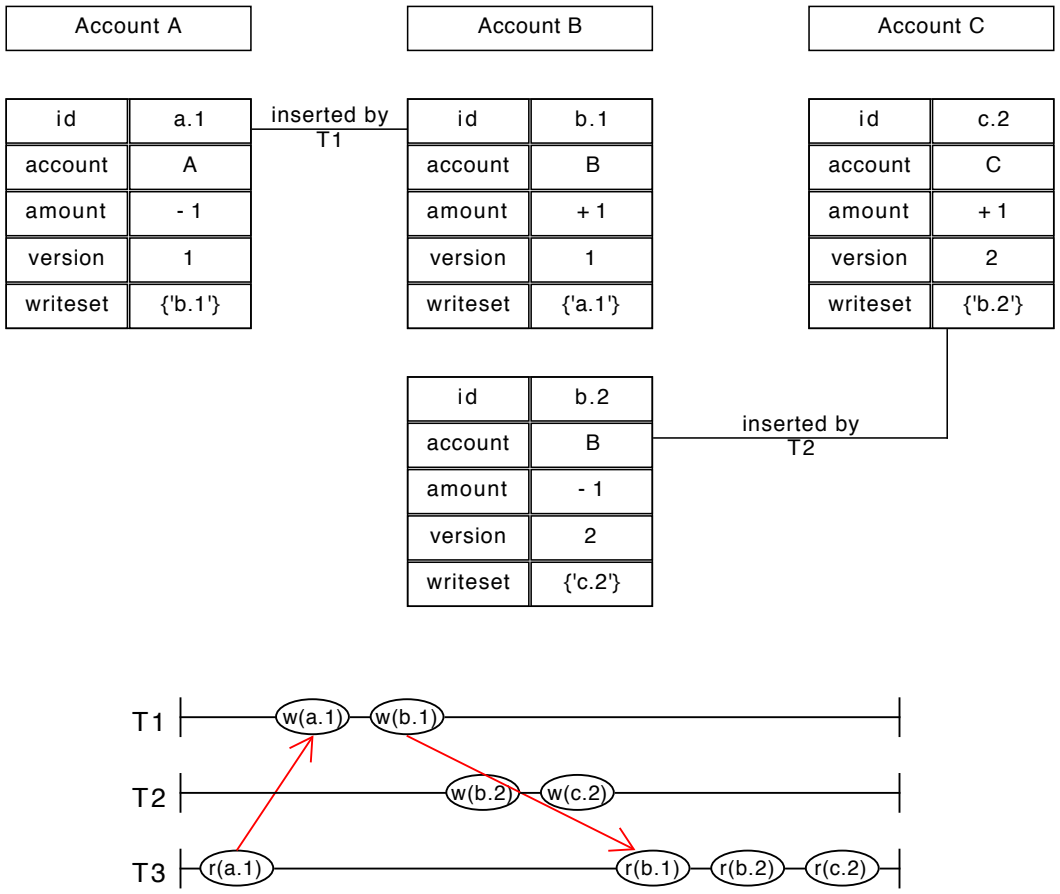


Figure 6.8: Example of data model using transfer records to avoid hidden fractured reads

Instead of using a scalar transaction id, "a vector clock with one entry per client in the system" [BFH+] could be used to track causality between transactions of different users. However, the incurred overhead of this approach appears too high for practical use.

Without the protocol extension RAMP does guarantee serializability for all use cases meeting the so-called *read-subset-items-written (RSIW) property*. This means that "for read-only and write-only transactions, if each reading transaction only reads a subset of the items that another write-only transaction wrote, then RA isolation is equivalent to [...] serializable isolation" [BFH+]. The rule of thumb for RAMP is: *Items that are read together should be written together*.

Because of the significant use case and data model restriction and the complexity of RA isolation, RAMP transactions will not be integrated into the public API for application developers but only be used for potential internal use cases in Orestes.

7 Evaluation

This chapter shows the evaluation results for certain aspects of DCAT. Section 7.1 covers the conflict rate and latency improvement due to caching, Section 7.2 shows measurements for transaction throughput and scalability and Section 7.3 describes a real-world application benefitting from the performance of DCAT.

7.1 Conflict Rates and Latency

This section evaluates the conflict rate and latency improvements due to caching. To this end, the conflict rates and latency of DCAT are compared to the ones of optimistic transactions without caching to verify the hypothesis from Subsection 4.6.2: “Caching improves the abort rate of transactions with optimistic concurrency control substantially”.

7.1.1 Experimental Setup

As discussed in Subsection 4.6.1, it is way too difficult to calculate the expected conflict rates for cached transactions. On the other hand, it is too expensive to execute the needed measurement workloads in a real-world deployment with CDN and cloud deployment. The cost effective and reliable way is to measure the conflict rates and latency in a Monte Carlo [Moo97] simulation that models the whole client-server architecture including cache hierarchy and transaction processing.

Simulation

The simulation models the client-server architecture of Orestes including the caching hierarchy and tunable latencies between the components. The main components of the simulation are depicted in Figure 7.1. The execution of generated workloads is controlled by schedulers that either maintain a given number of concurrent executions (maximizing the throughput for a given concurrency) or execute operations at a given rate (keeping a static load on the system).

The schedulers draw operations from workload generators and assign them to the client with the least queued operations. The operations are executed by the client using

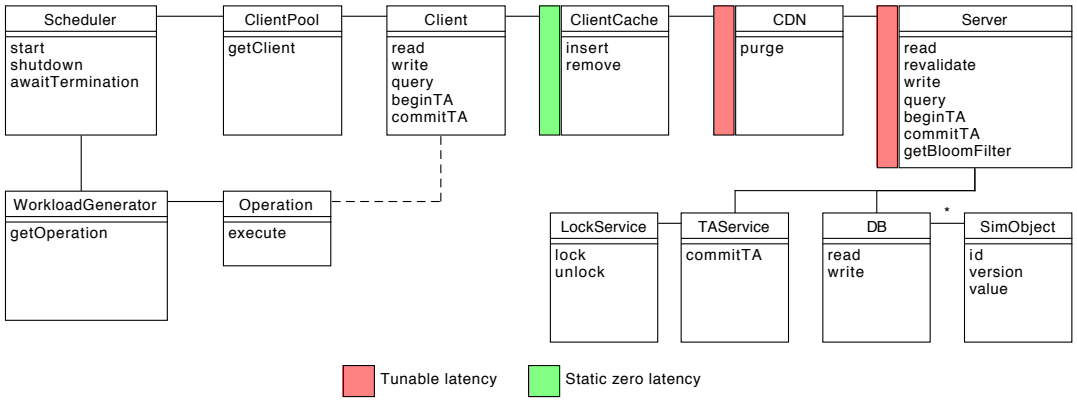


Figure 7.1: Main simulation components

an available connection (load balancing over 6 connections, similarly to web browsers). The operations are passed through and possibly handled by the client- and CDN cache, which also can be deactivated. At last instance operations are handled by the server that utilizes the database, a transaction service for commit handling and a lock service for coordination. Latencies are simulated with normal distributions between clients, caches and server and can be tuned to evaluate various setups. The default mean latency setting representative for common web applications [GSW+16] and used for all following simulation runs are: client to CDN $4\,ms$, CDN to server $73\,ms$, server to CDN (for invalidations) $100\,ms$. As a consequence, a read request with a cache miss needs $4+73+73+4 = 154\,ms$. A CDN cache hit only needs $8\,ms$ and a client cache hit is instantaneous.

Workloads

The simulation workloads are designed to measure the transaction latency and conflict rate as a function of the transaction size. All workloads are executed on a set of 10000 object and distributed over 25 clients, each with 6 connections. The baseline system traffic is generated by two schedulers. One executes a number of write operations per second to keep a static update rate on the objects. The other executes read operations to fill the caches. This baseline traffic simulates the usage of the web application.

On top of this baseline traffic a third scheduler executes read-only transactions of different sizes, i.e. transactions that load different numbers of objects. Read-only transactions are chosen to keep the static update rate generated by the baseline traffic[1] and because

[1] Write operations inside the transactions would effect the update rate depending on their abort probability, which would corrupt the common baseline for transactions with and without caching.

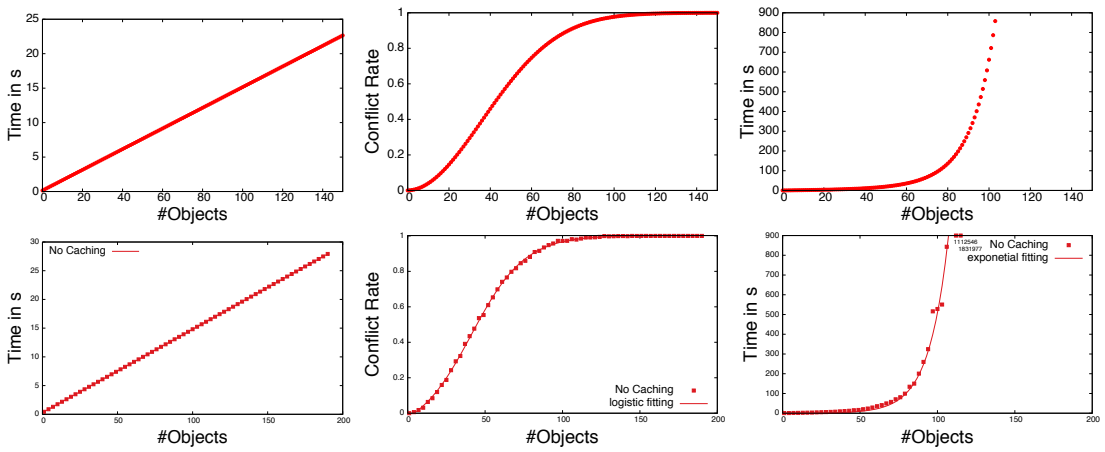


Figure 7.2: Theoretical results (top) compared to the simulation results (bottom) for transactions without caching

transaction aborts only depend on read objects anyway. These read-only transactions provide the measurements for latency and abort rates as a function of transaction size.

Validation

To cross validate the simulation concept and implementation with the theoretical analysis of transactions without caching from Subsection 4.6.1, Figure 7.2 compares the theoretical to the simulation results (without caching) for the same baseline of 50 updates per second and a uniform read distribution.

Figure 7.2 compares latency (a single execution of the transaction, whether aborted or committed) and conflict rate of transactions as well as their total runtime including all retries until success (called retried runtime), for transactions from 1 to 190 read objects. The sampling of the simulation uses transaction sizes in steps of three $(1, 4, 7, \dots)$. According to the figures, the simulation results match the theoretical analysis, which verifies the simulation implementation and the chosen parameters.

7.1.2 Experimental Results and Interpretation

The following comparisons between simulations results for transactions with and without caching show the latency and conflict rate improvements and verify the hypothesis. All comparisons depict the transaction latency, the conflict rate and the retried runtime as functions of the transaction size.

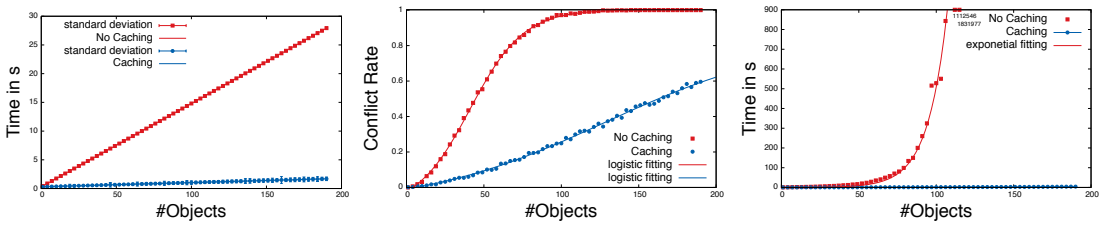


Figure 7.3: Simulation results of transactions with uniformly distributed reads and writes.

The first comparison – depicted in Figure 7.3 – shows the results for a baseline of 50 writes and 1500 reads per second ($\approx 95\%$ reads) with uniform distribution over the objects.

- The first graph shows the immense benefit caching has on transaction latency. While standard transactions need $\approx 150\,ms$ for each read operation, cached transactions are with $10\,ms$ per read on average 15 times faster.
- This latency benefit has a substantial effect on the transaction abort rate as shown in the second graph. Transaction conflict rates have two main drivers: the number of read objects and the time between read and commit. The latency improvement decreases the second driver and therefor improves the transaction abort rate substantially.
- The practical effect of the conflict rate improvement becomes clear by visualizing the expected total runtime of a transaction including all retries (in the third graph). Standard transactions with only 10 objects need 2 seconds, with 75 even 100 seconds and with more than 100 objects they practically never commit. Cached transactions on the other hand need only $400\,ms$ when loading 10 object and can contain 70 objects before hitting 2 seconds of runtime. In summary the advantage for retried runtime of cached transactions is not only the 15 time latency reduction but grows more than exponentially with the transaction size.

Object Distribution

Transaction conflict rates depend on the update rates of the loaded objects, which were the same for all of the 1000 objects so far due to the uniform distribution of baseline

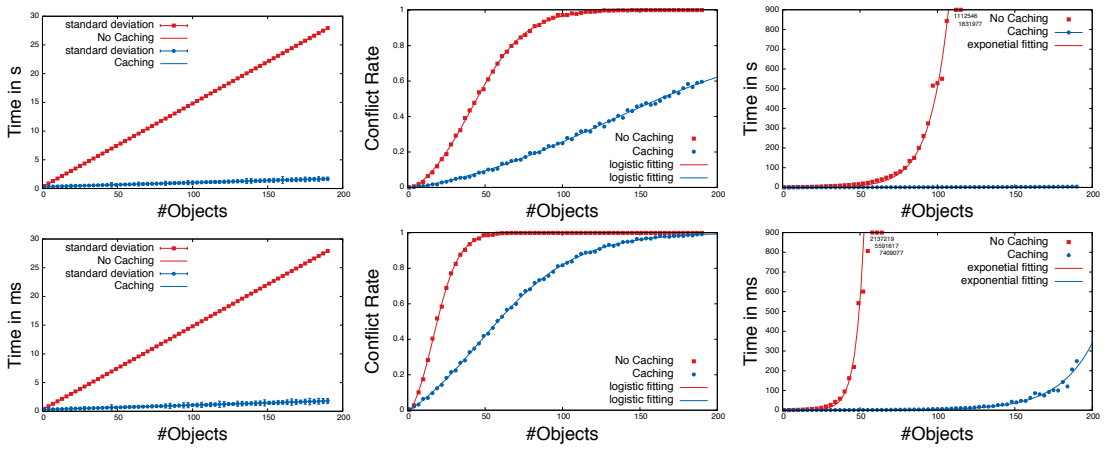


Figure 7.4: Simulations results for uniformly (top) and Zipfian (bottom) distributed reads and writes

read and write operations. The popularity of objects in workloads common for web applications is ,however, not a uniform distribution but rather Zipfian [BCF+99], e.g. the most popular object is twice as popular as the second and so forth. To capture these more relevant workloads Figure 7.4 compares the simulation results with a uniform object distribution to the results with a Zipfian ($s = 0.6$) distribution. The distribution is used to draw objects for the baseline read and write traffic and also for the read-only transactions.

- The transaction latency – depicted in the first graph – barely changes with the Zipfian distribution. Only the variance for cached transactions increases slightly.

- Unlike the latency, the conflict rate (second graph) for both cached and non-cached transactions changes significantly with the Zipfian distribution. This is explained by the fact that the most popular objects, which are loaded in most transactions, have a much higher update rate than before and thus lead to more transaction aborts. Nevertheless, the results show that caching still significantly decrease transaction conflict rates.

- Again, the third graph shows the practical effect of decreased conflict rates on the runtime of retried transactions. Non-cached transactions with more than 50 objects practically never commit while the runtime of cached transactions can contain 175 objects before hitting 100 seconds of runtime.

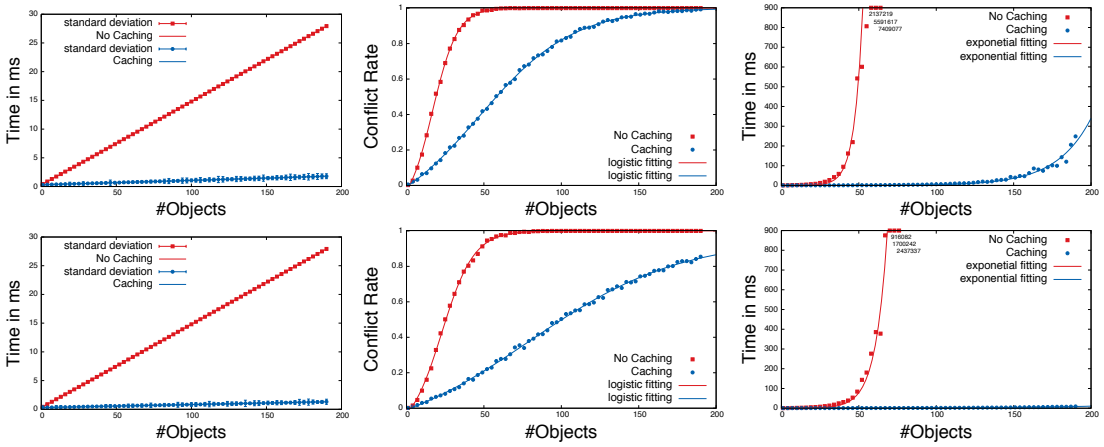


Figure 7.5: Simulation results of transactions with Zipfian distributed reads and writes. 50 writes per second on the top, 25 writes per second on the bottom

Update Rates

Another parameter that changes object update rates is the number of baseline writes per second. Results of decreasing this update rate from 50 to 25 writes per second ($\approx 98\%$ reads) while keeping the Zipfian object distribution are depicted in Figure 7.5.

- As shown in the first graph, the transaction latency slightly decreases with fewer writes because cache hit probability increases.
- The transaction conflict rate – shown in the second graph – decreases significantly, especially for cached transactions due to fewer stale reads from caches.
- Yet again, the third graph shows the practical effect of decreased conflict rates on the runtime of retried transactions. The runtime of cached transaction only slightly increases in the depicted transaction size window, while non-cached transactions with more than 50 reads are practically unusable.

7.1.3 Not Analyzed Parameters

There are still some parameters whose effects on latency and conflict rates could be analyzed. Most important are:

1. Different latencies between clients, caches and server. The speedup of cached transactions is determined by the cache hit rate and the latency differences of the client-cache and the client-server connection. Different latency parameters could simulate

various geographical locations of clients and estimate the impact of caching for transaction latency and conflict rates. These figures are especially interesting for applications with geographically distributed customers.

2. Different TTL estimation strategies. Choosing the right TTL has a high impact on cache hit rates as found in [GSW+16], because on the one hand with an underestimated TTL objects are removed from caches although still valid. On the other hand, with an overestimated TTL stale objects remain cached and must be inserted into the Bloom filter. While the Bloom filter ensures that stale objects are revalidated by each client, the resulting up-to-date cache entry can only be used once the object is removed from the Bloom filter again, which depends on the TTL.

   The current Orestes implementation as well as the simulation estimate the ideal TTL for each object returned by the server based on the current update rate. There are ,however, other strategies for choosing the TTL, based on quantiles for example, which could make a significant difference for cache hit rates and thus transaction latency and conflict rates.

Furthermore the simulation loads the Bloom filter at transaction begin. This reduces the probability of stale cache reads but needs an additional round-trip. Further simulations could evaluate under which circumstances the elimination of this roundtrip would lead to an increased conflict rate.

7.2 Throughput and Scalability

This section evaluates the overhead of the transaction processing compared to standard read and write operations in terms of latency and throughput. Furthermore it attempts to show the linear scalability of the transaction processing and analyzes the capacity of the Redis coordinator implementation, i.e. at which throughput the Redis coordinator becomes a bottleneck.

7.2.1 Experimental Setup

Cluster

This evaluation step is concerned with throughput and scalability of the transaction processing in the backend and therefore ignores caching and real-world latencies between

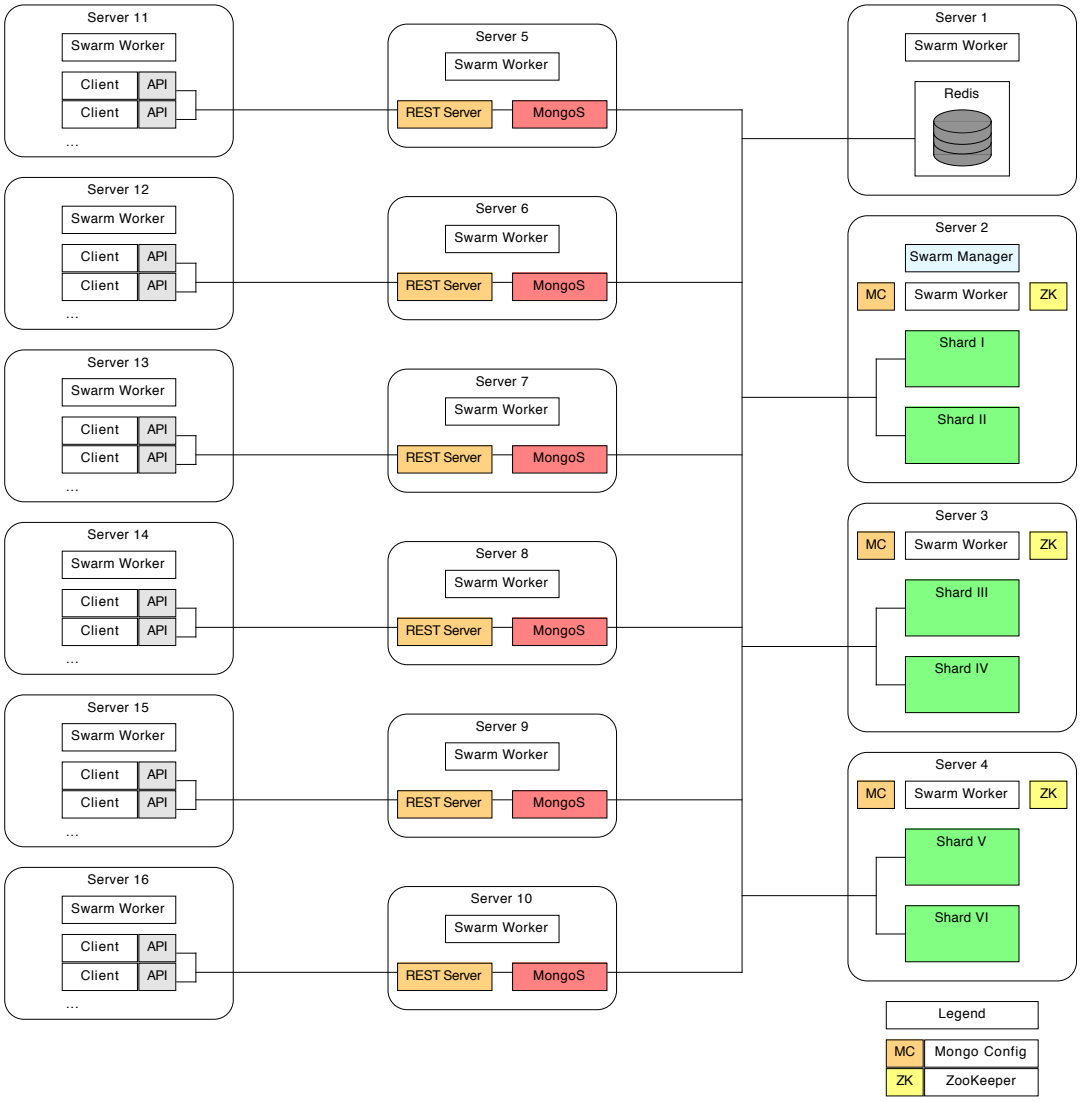


Figure 7.6: The cluster setup for the throughput and scalability measurements

client and server. The experimental setup consists of a number of benchmark clients[2], which execute transaction workloads against a cluster of REST servers backed up by a sharded MongoDB database and a single Redis coordinator instance.

Figure 7.6 shows the setup of 16 servers hosted on IBM's SoftLayer cloud, each with a 4-core CPU, $2.0\,GHz$ and $16\,GHz$ main memory, managed using Docker Swarm with ZooKeeper [Doc, Zoo]. The first server executes the Redis coordinator while the next three host the MongoDB config servers and six MongoDB shards, which contain 10000 objects (uniformly distributed among the shards).

Each of the servers five to ten is equipped with a REST server and a MongoS (MongoDB load balancing proxy) and has another server associated with it, hosting the benchmark clients, which execute transaction workloads against the REST server.

Workloads

The evaluation strategy is to start with a single REST server deployment, scale up the benchmark clients until the maximum throughput is reached and than gradually scale up the number of REST servers from one to six and the clients accordingly.

Each benchmark client executes 100 transactions per second (with a maximum of ten parallel transactions) and thus simulates 100 real-world clients executing one transaction per second. Each transaction reads ten objects and updates two of them. The same workload is executed is a non-transaction environment as a baseline for comparison.

7.2.2 Experimental Results and Interpretation

Throughput and Processing Overhead

The throughput results of scaling up clients for one and two REST servers are shown in Figure 7.7. The transactional throughput peaks at 589 transactions per second for one REST server and 1072 for two, whereas the non-transactional execution peaks at 666 for one REST server and 1238 for two. The resulting overhead of the transaction processing can be estimated at approximately 12%.

Figure 7.8 shows the latency degradation for one and two REST servers. Both, the transactional and the non-transactional execution start at $15\,ms$ of latency on a server with little load. The latency increases slowly until a throughput of 500 transactions per second (or 900 receptively) at which the exponential increase becomes visible. There is a significant latency overhead of the transactional execution of approximately 40%, which ,however, becomes negligible if the overall transaction latency is considered.

[2]The benchmark client and cluster setup manual can be found at `https://github.com/Baqend/transaction-performance-test`

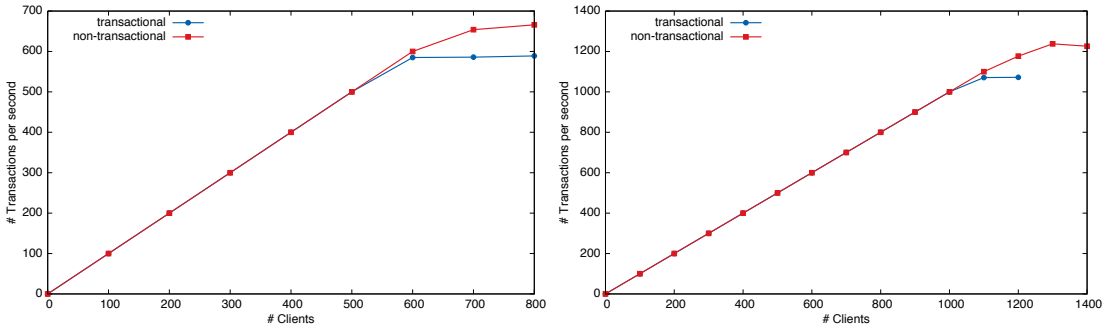


Figure 7.7: Throughput of one (left) and two (right) REST servers as a function of the number of clients

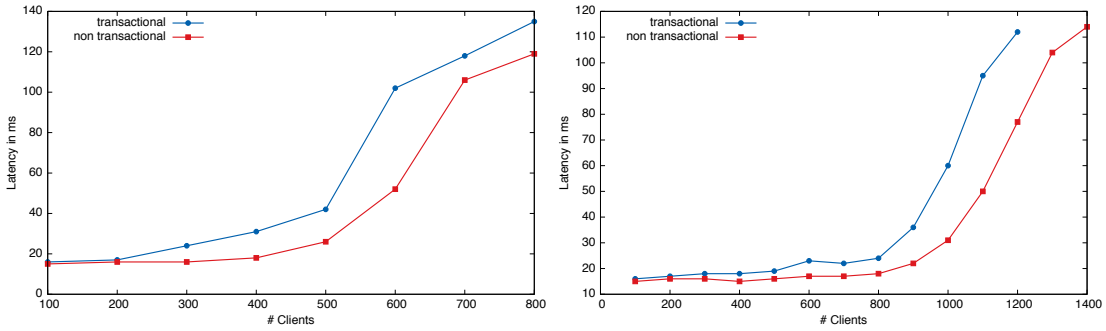


Figure 7.8: Latency of a transaction for one (left) and two (right) REST servers degrading with the number of clients

Scalability

The expected scalability of the transaction processing is a peak throughput increasing linearly with the number of REST servers until the Redis coordinator implementation becomes a bottleneck and the throughput stops increasing. The non-transactional execution on the other hand is expected to scale linearly with the number of stateless REST servers[3].

Surprisingly the results show a different behavior. Neither the transaction processing nor the non-transactional processing on the stateless REST servers scale linearly: Figure 7.9 shows the throughput of all tested REST server setups for both the transactional and non-transactional execution and Figure 7.10 shows how the peak throughput scales with the number of REST servers.

[3]The MongoDB setup is expected to comfortably cope with the tested number of REST servers.

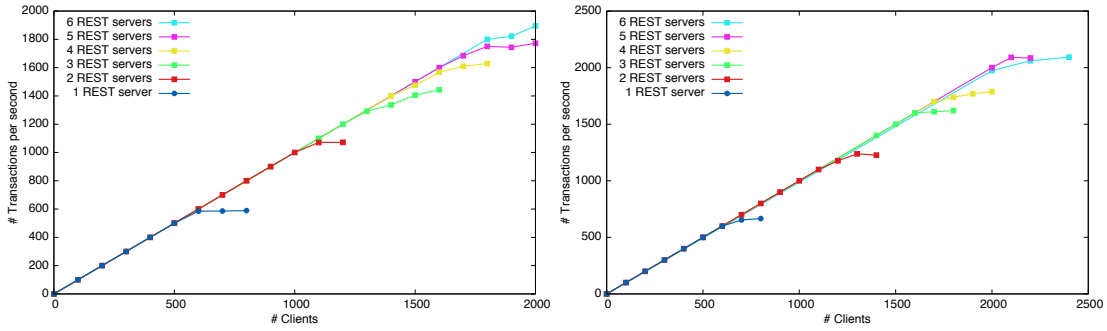


Figure 7.9: Throughput graphs of all REST server setups, transactional execution (left) non-transactional execution (right)

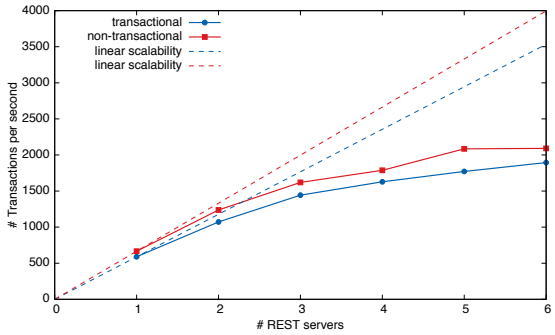


Figure 7.10: Scalability results for transactional and non-transactional execution

Figure 7.10 shows that the peak throughput of the transaction processing scales in the same way as the non-transactional execution, keeping a static overhead. The servers' resource utilization showed no indication of a bottleneck. The most apparent explanation of the suboptimal scalability would be that MongoDB uses write locks of a coarse granularity ([Mona]) for `findAndModify`-Queries even though these would not be needed. Further analyses are needed to find and resolve the problem that limits scalability. After solving this issue, the transaction processing is expected to scale linearly until the Redis coordinator becomes the bottleneck. However, both of these tasks are out of the scope of this thesis due to deployment costs (≈ 200 € per day) and time limitations.

Based on the observed resource utilization of the Redis coordinator of $\approx 50\%$ for 1900 transactions per second and assuming a near to linear scalability of the transaction processing, the coordinator is expected to become a bottleneck for peak throughput of 6 to 8 Orestes servers, i.e. 3500 to 4500 transactions (consisting of 10 read and 2 writes) per second.

7.2.3 Aspects Not Analyzed

There are still some aspects and effects to be analyzed in future work. Most important are:

1. Different transaction sizes and compositions. The throughput measurements cover only transactions of a static size and composition (10 read and 2 writes). It would be interesting how exactly the size and composition of the transactions would effect the throughput.

2. Caching for read operations. The experimental setup did not use caching and only measured the throughput of the REST server implementation. It is to be expected that read operations served by caches would decrease the server load and thus improve throughput and latency degradation significantly.

3. Sharding of the coordinator. After the single node Redis coordinator bottleneck threshold is identified, the scalability of a sharded coordinator and its latency effects should be analyzed.

7.3 Real-World Application

This chapter presents a real-world application for DCAT transactions to show both the need for distributed scalable transactions and the applicability of DCAT to use cases with strong requirements.

7.3.1 Use Case and Requirements

The dCache[4] project provides a distributed filesystem for large amounts of data, which is deployed in 74 places around the word including the DESY and CERN hadron collider and stores 114 petabytes of data. The filesystem is divided into multiple components including a metadata store among others. This metadata store – developed at DESY in Hamburg – is used to store file and directory metadata and navigate through the file system with full posix[5] semantics. The actual files are stored separately.

[4] `https://www.dcache.org`
[5] `http://de.wikipedia.org/wiki/POSIX`

The current implementation uses PostgreSQL and hits the throughput limit at ≈ 1000 transactions per second, which is prospectively insufficient for some dCache users. This challenge gave rise to the idea of an implementation based on Orestes and DCAT. The prototype based on a data model designed during this study is currently under development at DESY. The remainder of this section will present the system requirements and the data model with its design decisions.

Requirements

The metadata to store for files and directories include following attributes among others:

- **type**: The type of the object – file or directory.
- **creation**: The creation time of the object.
- **modification**: The last modification time.
- **change**: The last modification of the metadata.
- **access**: The time of the last access of the object.
- **acls**: The access rights of the object.

The operations are based on the posix semantics and include the following among others:

- `stats(path)`: Returns the specified object's metadata.
- `update(path, attributes)`: Updates the metadata of the specified object and sets the change time.
- `create(path, o_type, attributes)`: Creates an object of the given type with the given attributes and updates the modification time of the parent directory.
- `rename(old_path, new_path)`: Renames the object associated with the old path and moves it to the new path (including all underlying objects), both parent directories' modification times are updated.
- `link(obj_id, parent_id, name)`: Creates a reference (hard link) with the given name in the specified parent directory to the specified object.

Most importantly, the system must guarantee that no two objects with the same path can exist in the filesystem.

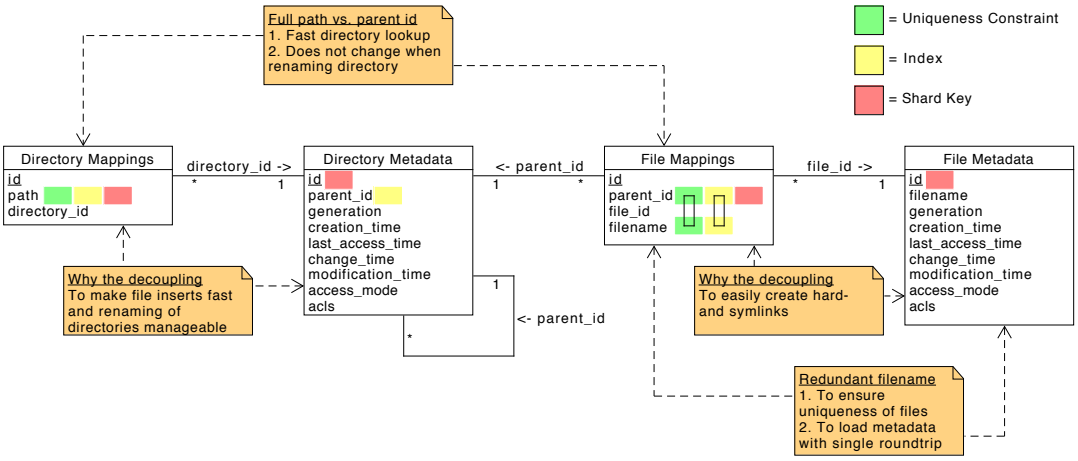


Figure 7.11: The proposed data model for the dCache metadata store implementation as a UML diagram

All of the operations described above are heavily used in the diverse landscape of dCache deployments and need to have low latency, high throughput and a small abort rate. Further system properties important for the data model are that the system exhibits:

- high numbers of file (≥ 1000 per second) and directory inserts.
- rather flat (but wide) directory trees (≈ 10 directories deep).
- frequent rename operations that also move whole subtrees.

In essence the new system should scale with the number of files and directories and should keep the complexity of operations at $\mathcal{O}(1)$.

7.3.2 Data Model

The proposed data model, depicted in Figure 7.11, is based on the requirements of the metadata store and takes the pragmatics of DCAT transactions into account.

The data model consists of four entities: directory mappings that each map a directory id to a path, directory metadata saving the actual metadata information and similar entities for files. Each of the entities has a shard key for horizontal distribution and indexes where needed for faster access. The directory path as well as the combination of file name and parent id have a unique constraint to prevent duplicate entries in the file system.

Design Decision

The reason to separate the location information from the rest of the directory metadata was to make inserting files fast and at the same time renaming (or moving) of directories possible. With this separation, a file insert operation can partially update the modification time (with the $\$max$ operator) while the rename operation can load and update the directory mapping to move or rename the directory with a much lower abort rate. With a single directory metadata object, the rename of a directory (that has to move all subdirectories) would exhibit a high abort rate making it very slow or even impossible under high contention. The reason to also separate the file location and other metadata is to easily create hard- and symlinks by inserting another file mapping and to easily move files to another directory.

File and directory mappings differ in a fundamental way. While the later stores the full path to the directory, the former only stores the id of the parent directory. The advantage of storing the full path is that path-based lookups only load a single mapping instead of heaving to load all directories on a path which would make file inserts into a directory very slow. The advantage of storing references to parent directories is that rename (move) operations are very fast because only the reference changes. The proposed data model tries to combine these advantages. On the one hand, directories can easily be found in one lookup and files only need two (first the directory then the file). On the other hand, files can easily be moved to another directory while for directories every subdirectory must be updated too.

A rather minor design decision is to store the file name in two places. It is needed in the file mapping to prevent duplicates, but is also included in the file metadata in order to load the whole metadata in one step if the file id is known.

Operations

The following three examples illustrate how the required operations are performed on the proposed data model.

Loading the Metadata:
`stat(path)` (no transaction required)

1. Load entry from the directory mappings using the path.
2. If it does exist, resolve the directory metadata using the id.
3. If it does not exist, it might be a file path. In this case split the path in parent path and file name and load the parent directory mapping.

4. Load the file mapping with the found parent directory id and the file name.
5. Finally, resolve the file metadata with the file id.

All involved objects can be cached in the client. Thus, optimal round-trip count is 0. The worst case for directories is 2 and for files 4.

Inserting Files:
`create(path, file, attributes)` (as transaction)

1. Load the parent directory mapping.
2. Insert the new file metadata entry.
3. Insert the file mapping.
4. Partially update the parent directory's modification time.

The optimal round-trip count is 2 (transaction begin and commit). The worst case is 3.

Renaming Objects:
`rename(old_path, new_path)`

1. Load entry from the directory mappings using the old path.
2. If it does exist, it is a folder:
   a) Execute a prefix query on the directory mapping using the old path.
   b) Rename the path of all found mappings.
   c) Resolve old and new parent directories and partially update their modification time.
3. If it does not exist, it is a file:
   a) Find the file mapping.
   b) Resolve the new parent directory id
   c) Update file mapping with the new parent, partially update the new and old parent directory and if required rename the file.

The optimal round-trip count for is 3 (worst case 5).

Summarized Achievements and Missing Features

The proposed data model achieves the required properties of efficient sharding (high scalability), low latency, operations in $\mathcal{O}(1)$ (except for renames and deletes) and low conflict rates for the most relevant operations.

The implementation and usage of the proposed data model, however, require some extensions to Orestes and the transaction processing.

1. Efficient cache lookups using unique keys instead of primary keys to get directory mappings from cache based on the path.
2. The enforcement of unique constraints by the transaction processing (as mentioned in Subsection 4.7.1) to prevent duplicates in the file system.
3. The handling of phantom problems in the transaction processing (as mentioned in Subsection 4.7.2) to prevent inconsistencies during renames.

The prototype currently under development at DESY will showcase the usability and performance of DCAT.

8 Related Work

The increasing importance of storage requirements like scalability, availability and low latency imposed by modern data driven web applications drives the development of a whole landscape of NoSQL databases, each with its own set of tradeoffs. Most of these databases sacrifices features known from classical relational databases to meet the new requirements – first and foremost the feature of ACID transactions.

These requirements are, however, in conflict with the need of interactive web applications for strong consistency and a rich set of features to achieve an intuitive user experience and a short time to market. In balancing these requirements a lot of systems and strategies for distributed transactions have been developed with the focus on retaining scalability and availability as much as possible.

The following transactional systems can best be distinguished by the scope of their transaction support (single-shard or multi-shard transactions), the achieved isolation level, used concurrency control mechanism and the used commit protocol.

Google was one of the first to react on their customers need for transactional operations with the development of Megastore [BBC+11], which is based on Google's BigTable [CDG+08] and "blends the scalability of a NoSQL datastore with the convenience of a traditional RDBMS" [BBC+11]. Megastore synchronously replicates data over wide area networks using Paxos [L+01] and shards data across fine-grained partitions named *entity groups*. Transactions are only supported within an entity group obviating the need for a distributed commit protocol. The protocol uses optimistic concurrency control on entity group level. The main problem of Megastore that is used in Google App Engine [Ciu09], beside the limited transaction support, is the performance bottleneck of only a few write operations per second per entity group.

Similarly to the idea of entity groups, G-Store [DAEA10] supports ACID transactions on key groups, which are (in contrast to entity groups) not static sets but can be dynamically created and dissolved using a protocol similar to the Two-Phase-Commit protocol (2PC) [Lec09]. Each key group has a master key whose server manages the transaction as in centralized database using optimistic concurrency control, assuming low contention per key group. The approach works best if key groups have a long life time and low contention.

Motivated by the poor write performance of Megastore, Google developed Spanner [CDE+13], which is also based on BigTable and global synchronous replication with Paxos. Spanner supports both general-purpose transactions with 2PC and pessimistic concurrency control per partition as well as non-locking read-only transactions with snapshot isolation using transaction begin- and commit timestamps and the multi-version support of BigTable. The implementation of Spanner highly relies on Google's TrueTime API built from expensive GPS servers and atomic clocks to minimize clock drifts for transaction timestamps. Spanner is the core of Google's distributed RDBMS F1, which powers their advertisement business.

A third transactional system built by Google and based on BigTable is Percolator [PD10], which maintains Google's web search index by processing small updates to this huge dataset. Percolator supports snapshot isolation by using a timestamp oracle and a lightweight lock service to delay read operation if writes are in progress. Transactions are committed using 2PC and optimistic concurrency control on partitions by comparing versions similar to DCAT. The transaction processing, however, is throughput oriented – matching the requirements of Google's search index – and therefore exhibits high latencies, inappropriate for OLTP workloads.

MDCC [KPF+13a] for geo-replicated datastores on the other hand focuses on reducing round-trips and therefore latency by using Multi-Paxos [Lam98] for replication and transaction commit. With a Paxos instance per object to agree on a commit decision decided locally using optimistic validation, MDCC supports read-committed isolation. It can, however, be extended to higher isolation levels. In terms of throughput and latency MDCC outperforms Megastore and Spanner significantly. The latency copared to DCAT, however, is rather high, due to geo-replication round-trip latencies.

Especially targeted towards web applications is ElasTraS [DEAA09], an elastic datastore matching its cloud environment. ACID transaction are, however, only supported within partitions using traditional concurrency control. Across partitions only so-called mini-transactions are supported, which ensure durability but not isolation.

CloudTPS [WPC12] is a transaction management system also targeting web applications in the cloud, which supports general-purpose transactions using 2PC for transaction commit and timestamp ordering for concurrency control. CloudTPS is deployed as a middleware between web application and storage system and thus abstracts from the actually used database similar to DCAT. The system uses Local Transaction Managers (LTMs), which partition the application data (independent of the underlying storage system) and work on their own copy of the data replicated from the storage system. For each transaction one LTM is chosen to coordinate the transaction executed on several LTMs.

The library Cherry Garcia [DFR15] even supports distributed transactions across heterogeneous stores, enabling polyglot persistence as DCAT. Supported stores need linearizable read operation, conditional updates and multi-version support. Currently Windows Azure Storage [CWO+11] and Google Cloud Storage [Goo] are supported. The system uses an "appropriate" source of timestamps (like the TrueTime API) to assign transaction begin- and commit timestamps to enforce snapshot isolation. Read operations use these timestamps and linearizable read operations to read from the transaction's snapshot. Writes are buffered at the client side and committed using optimistic validation and a commit protocol that atomically exposes writes prepared during the validation.

Another system that can be implemented on top of various stores is Omid [KPF+13b], which implements lock-free transactions with snapshot isolation. Instead of a distributed protocol, a centralized so-called status oracle (SO) is used, which assigns begin- and commit timestamps, ensures that transactions read of snapshots and validates the writes at transaction commit. Even though the snapshot insurance can be executed at the client (by replicating status information from the SO to the client), the serial processing of transaction commits on the single SO instance imposes a bottleneck.

Calvin [TDW+12] also runs alongside non-transactional storage systems to enable transactional access. To this end, transactions are first written to a shared log (replicated using Paxos), then schedulers agree on a serialization and execute transactions potentially in parallel. Shortcomings of this approach are a high transaction latency and that only prepared transactions can be executed and no interactive ones.

The redesigned distributed RDBMS H-Store [KKN+08] (which has evolved to VoltDB [SW13]) is rather part of the NewSQL than NoSQL landscape. It has the same limitation to pre-defined stored procedures for transactional access as Calvin. A lazy execution planner decides on a transaction's execution plan and the transaction manager uses a generic distributed transaction framework (using both pessimistic and optimistic concurrency control) to coordinate the execution.

Another distributed SQL database is Cloud SQL Server [BCD+11] (used for SQL Azure [Micb] and Microsoft's Exchange Hosted Archive [Micc]). It avoids the use of 2PC by restricting transactions to a single partition using classical locking for serializable isolation.

The new storage engine of Microsoft SQL Server [Gra97], Hekaton [DFI+13] is optimized for main memory storage systems with high concurrency on a single machine and thus not distributed, but has nevertheless interesting transaction processing. It uses optimistic multi-version concurrency control for ACID transactions similar to snapshot isolation implementations but with the difference that read objects are validated at transaction commit like in DCAT. In contrast to DCAT, the validation phase is isolated from write operations overlapping with the read set. Instead, if a transaction reads a non-

committed value, a commit dependency between these transactions is tracked, meaning the reading transaction can only commit if the transaction that has written the value commits. While improving the concurrency of the transaction processing, this approach leads to cascading aborts and potentially long wait times for transactions.

The last system in this overview is FaRMville [DNN+15], a distributed datastore based on persistent main memory hardware to achieve very short latencies and high throughputs. It is mentioned despite its special hardware requirements – inappropriate for most web applications – because its commit processing works similar the same as DCAT's. The commit protocol acquires locks for the involved objects, validates the read version are still up-to-date, logs the commit, writes the objects to their respective partitions and removes the commit logs once changes are persistent. Read operations are indeed not served by caches but FaRMville buffers writes locally until commit, just like in DCAT. The performance figures with 4.5 million TPC-C *new order* transactions ([TPC]) per second are impressive.

Conclusion

DCAT achieves multi-partition ACID transactions with serializable isolation and has a lot of similarities to some of the presented systems:

- It is designed for polyglot persistence (like Cherry Garcia and similar to CloudTPS, Omid and Calvin).
- It buffers write operations at the client side until commit (like Cherry Garcia and FaRMville).
- It uses optimistic concurrency control (like Megastore, G-Store, Percolator, MDCC, Cherry Garcia, H-Store, Hekaton and FaRMville).
- It uses a protocol quite similar to 2PC (used in Spanner, Percolator, G-Store, CloudTPS and Cloud SQL Server), where instead of distributing the validation process across the shards as in 2PC, the information needed for the validation decision is gathered for a central commit decision.

It also shares the target of reducing read latency with geo-replicated systems like Megastore, Spanner and MDCC but uses caching instead of geo-replication to bring data closer to clients.

This leads to fundamental differences. To our knowledge DCAT is the only transaction concept that uses web caching to reduce transaction latency and decrease the problem of conflict rates associated with optimistic concurrency control. The caching-approach to low

latency transactions is not only new compared to geo-replication, it is also significantly more effective, because the commit processing is not slowed down by global round-trips for replication and agreeing on a commit decision.

9 Conclusion

9.1 Future Work

There are a lot of tasks remaining for future work, both in extending the transaction concept and in evaluation. The most relevant tasks sorted roughly by their importance are the following.

1. Fix the scalability problem identified in the Orestes cluster setup (see Section 7.2) and repeat the scalability experiments to verify the expected Redis coordinator bottleneck threshold.

2. Extend the transaction processing with the support for consistency constraints (most importantly a unique constraint) and design a user interface to specify them.

3. Extend the transaction protocol with the support for queries and expand the concurrency control to avoid the phantom problem.

4. Design and implement a comprehensive, continuous evaluation for the transaction's availability and recovery mechanism based on Netflix's *Chaos Monkey* approach [BT].

5. Evaluate missing aspects of the throughput and scalability evaluation mentioned in Subsection 7.2.3 as well as of the conflict rate evaluation mentioned in Subsection 7.1.3.

6. Evaluate the dCache metadata store prototype (and potentially the final implementation) once it is ready.

7. Design and implement a sharded (and potentially replicated) coordinator and evaluate the scalability (and potentially the availability) of the transaction processing again.

8. Implement fine-grained locking for partial updates, refactor the recovery mechanism accordingly and evaluate the throughput and latency benefits.

9. Implement the TPC-C benchmark [TPC] to get results comparable to those of competitors for both throughput and cost-effectiveness.

9.2 Summary

This study proposed a concept for scalable distributed cache-aware transactions (DCAT) based on the analysis of web application requirements and distributed transaction theory and discussed its database independent implementation in the Backend-as-a-Service Orestes.

The transaction concept of DCAT implements ACID semantics with COCSR isolation and is optimized for low latency using web-caching for read operations as well as a client-local workspace to buffer writes. Elastic scalability can be achieved by varying the number of REST servers within Orestes and sharding the lock-service. Fault tolerance is implemented in terms of an automatic recovery mechanism. These properties are achieved while only relying on a generic CRUD interface with a linearizable read operation for the underlying storage system.

The presented evaluation shows the low overhead of the implementation, the high potential for scalability and especially the huge latency benefits achieved by caching. Furthermore, a first prototype of the dCache metadata store is being implemented at DESY, which showcases the concept's applicability and data modeling pragmatics.

At last, the evaluation confirmed the hypothesis that caching significantly alleviates the problem of high abort rates introduced by the optimistic concurrency control. The abort rate problem is furthermore compensated by the two presented optimizations: partial updates for write operations and RAMP transactions for read-only transactions. With the key observation from MDCC that "[in] real workloads [...] either conflicts are rare, or many updates commute up to a limit (e.g., add/subtract with a value constraint that the stock should be at least 0)". [KPF+13a] It can be concluded that DCAT is a great fit for data-driven web applications and therefore for Orestes.

Bibliography

[Ady99] Atul Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

[Arc16] HTTP Archive. Stats. `http://httparchive.org/interesting.php`, Feburary 2016. [Online; accessed 03-Feburary-1016].

[BBC+11] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, volume 11, pages 223–234, 2011.

[BCD+11] Philip A Bernstein, Istvan Cseri, Nishant Dani, Nigel Ellis, Ajay Kalhan, Gopal Kakivaya, David B Lomet, Ramesh Manne, Lev Novik, and Tomas Talius. Adapting microsoft sql server for cloud computing. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1255–1263. IEEE, 2011.

[BCF+99] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 126–134. IEEE, 1999.

[BFH+] Peter Bailis, Alan Fekete, Joseph M Hellerstein, Ali Ghodsi, and Ion Stoica. Readrepair. `http://www.bailis.org/private/draft/ramp-tods-draft.pdf`. Not published.

[BFH+14] Peter Bailis, Alan Fekete, Joseph M Hellerstein, Ali Ghodsi, and Ion Stoica. Scalable atomic visibility with ramp transactions. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 27–38. ACM, 2014.

[Blo70] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[Bru09] Jake Brutlag. Speed matters for google web search. `http://venturebeat.com/wp-content/uploads/2009/11/delayexp.pdf`, June 2009. [Online; accessed 04-Feburary-2016].

[BT] Cory Bennett and Ariel Tseitlin. Chaos monkey released into the wild. `http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html`. [Online; accessed 2-Jun-2016].

[CDE+13] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[Ciu09] Eugene Ciurana. Google app engine. *Developing with Google App Engine*, pages 1–10, 2009.

[CRF08] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 729–738, New York, NY, USA, 2008. ACM.

[CWO+11] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, 2011.

[DAEA10] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 163–174. ACM, 2010.

[DEAA09] Sudipto Das, Amr El Abbadi, and Divyakant Agrawal. Elastras: An elastic transactional data store in the cloud. *HotCloud*, 9:131–142, 2009.

[DFI+13] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's

memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254. ACM, 2013.

[DFNR14] Anamika Dey, Alan Fekete, Raghunath Nambiar, and Uwe Rohm. Ycsb+t: Benchmarking web-scale transactional databases. In *Data Engineering Workshops (ICDEW), 2014 IEEE 30th International Conference on*, pages 223–230. IEEE, 2014.

[DFR15] A. Dey, A. Fekete, and U. Röhm. Scalable distributed transactions across heterogeneous stores. In *2015 IEEE 31st International Conference on Data Engineering*, pages 125–136, April 2015.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[DNN+15] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 54–70. ACM, 2015.

[Doc] Docker. Docker swarm. `https://www.docker.com/products/docker-swarm`. [Online; accessed 2-Jun-2016].

[FLO+05] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems (TODS)*, 30(2):492–528, 2005.

[GBR14] Felix Gessert, Florian Bucklers, and Norbert Ritter. Orestes: A scalable database-as-a-service architecture for low latency. In *Data Engineering Workshops (ICDEW), 2014 IEEE 30th International Conference on*, pages 215–222. IEEE, 2014.

[GFW+14] Felix Gessert, Steffen Friedrich, Wolfram Wingerath, Michael Schaarschmidt, and Norbert Ritter. Towards a scalable and unified rest api for cloud data stores. *Erhard Plödereder, Lars Grunske, Eric Schneider und Dominik Ull, Hrsg*, 44:723–734, 2014.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[Goo] Google. Google cloud storage documentation. `https://cloud.google.com/storage/docs/`. [Online; accessed 1-Jun-2016].

[Gra97] Jim Gray. Microsoft sql server. January 1997.

[Gri13] I. Grigorik. *High Performance Browser Networking: What every web developer should know about networking and web performance*. O'Reilly Media, 2013.

[Gro99] Network Working Group. Hypertext transfer protocol – http/1.1:. `https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html`, June 1999. [Online; accessed 04-Feburary-2016].

[GSW+15] Felix Gessert, Michael Schaarschmidt, Wolfram Wingerath, Steffen Friedrich, and Norbert Ritter. The cache sketch: Revisiting expiration-based caching in the age of cloud data management. *Datenbanksysteme für Business, Technologie und Web (BTW 2015)*, 2015.

[GSW+16] Felix Gessert, Michael Schaarschmidt, Wolfram Wingerath, Eiko Yoneki, and Norbert Ritter. Quaestor: Scalable and consistent query result caching on the web's infrastructure. *Submitted to SIGMOD 2017*, 2016.

[GTS+02] D. Gourley, B. Totty, M. Sayer, A. Aggarwal, and S. Reddy. *HTTP: The Definitive Guide: The Definitive Guide*. Definitive Guides. O'Reilly Media, 2002.

[Hof09] Todd Hoff. Latency is everywhere and it costs you sales - how to crush it. `http://highscalability.com/blog/2009/7/25/latency-is-everywhere-and-it-costs-you-sales-how-to-crush-it.html`, July 2009. [Online; accessed 04-Feburary-2016].

[HR13] T. Härder and E. Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Springer Berlin Heidelberg, 2013.

[KKN+08] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[Kle] Martin Kleppmann. How to do distributed locking. http://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html. [Online; accessed 022-March-2016].

[KPF+13a] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 113–126. ACM, 2013.

[KPF+13b] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 113–126. ACM, 2013.

[Kun] Gehrig Kunz. Readrepair. https://wiki.apache.org/cassandra/ReadRepair. [Online; accessed 17-May-2016].

[L+01] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[Lab] LabLua. The programming language lua. https://www.lua.org. [Online; accessed 9-July-2016].

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[Lec09] Jens Lechtenbörger. Two-phase commit protocol. In LING LIU and M.TAMER ÖZSU, editors, *Encyclopedia of Database Systems*, pages 3209–3213. Springer US, 2009.

[LFKA13] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger semantics for low-latency geo-replicated storage. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 313–328, 2013.

[Mica] Microsoft. Roll-forward recovery. https://msdn.microsoft.com/en-us/library/ms986590(v=exchg.65).aspx. [Online; accessed 2-Feb-2016].

[Micb] Microsoft. Sql-datenbank. https://azure.microsoft.com/de-de/services/sql-database/. [Online; accessed 1-Jun-2016].

[Micc] Microsoft. Übersicht über eha. https://technet.microsoft.com/de-de/library/gg191799.aspx. [Online; accessed 1-Jun-2016].

[Mona] MongoDB. Faq: Concurrency. https://docs.mongodb.com/v3.0/faq/concurrency/. [Online; accessed 27-May-2016].

[Monb] MongoDB. Query documents. https://docs.mongodb.com/manual/tutorial/query-documents/. [Online; accessed 9-July-2016].

[Monc] MongoDB. Update operators. https://docs.mongodb.com/v3.0/reference/operator/update/. [Online; accessed 19-May-2016].

[Mond] MongoDB. "partial object updates" will be an important nosql feature. http://blog.mongodb.org/post/307919034/partial-object-updates-will-be-an-important. [Online; accessed 18-May-2016].

[Moo97] Christopher Z Mooney. *Monte carlo simulation*, volume 116. Sage Publications, 1997.

[PD10] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, volume 10, pages 1–15, 2010.

[Rah88] Erhard Rahm. Optimistische synchronisationskonzepte in zentralisierten und verteilten datenbanksystemen/concepts for optimistic concurrency control in centralized and distributed database systems. *it-Information Technology*, 30(1):28–47, 1988.

[Reda] Redis. Distributed locks with redis. http://redis.io/topics/distlock. [Online; accessed 07-March-2016].

[Redb] Redis. Redis persistence. http://redis.io/topics/persistence. [Online; accessed 07-March-2016].

[San] Salvatore Sanfilippo. Is redlock safe? http://antirez.com/news/101. [Online; accessed 22-March-2016].

[Set81] Ravi Sethi. A model of concurrent database transactions. In *Foundations of Computer Science, 1981. SFCS'81. 22nd Annual Symposium on*, pages 175–184. IEEE, 1981.

[SGGS12] Abraham Silberschatz, Peter B Galvin, Greg Gagne, and A Silberschatz. *Operating system concepts*, volume 9. Addison-Wesley Reading, 2012.

[SGR15] Michael Schaarschmidt, Felix Gessert, and Norbert Ritter. Towards automated polyglot persistence. *atenbanksysteme für Business, Technologie und Web (BTW 2015)*, 2015.

[SPBZ11] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400. Springer, 2011.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys (CSUR)*, 37(1):42–81, 2005.

[SSS15] S. Sippu and E. Soisalon-Soininen. *Transaction Processing: Management of the Logical Database and its Underlying Physical Structure*. Data-Centric Systems and Applications. Springer International Publishing, 2015.

[SW13] Michael Stonebraker and Ariel Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[TDW+12] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[TPC] TPC. Tpc-c. `http://www.tpc.org/tpcc/default.asp`. [Online; accessed 1-Jun-2016].

[VP03] Athena Vakali and George Pallis. Content delivery networks: Status and trends. *Internet Computing, IEEE*, 7(6):68–74, 2003.

[VV15] Paolo Viotti and Marko Vukolić. Consistency in non-transactional distributed storage systems. *arXiv preprint arXiv:1512.00168*, 2015.

[Wes01] D. Wessels. *Web Caching*. O'Reilly Series. O'Reilly & Associates, 2001.

[WPC12] Zhou Wei, Guillaume Pierre, and Chi-Hung Chi. Cloudtps: Scalable transactions for web applications in the cloud. *Services Computing, IEEE Transactions on*, 5(4):525–539, 2012.

[WV02] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Series in Data Management Systems. Morgan Kaufmann, 2002.

[Zoo] ZooKeeper. Zookeeper: A distributed coordination service for distributed applications. `https://zookeeper.apache.org/doc/trunk/zookeeperOver.html`. [Online; accessed 07-March-2016].

[ZSS+15] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 263–278. ACM, 2015.

Acknowledgement

First, I would like to thank Felix Gessert for his outstanding supervision during the whole thesis. His exceptional expertise, supporting attitude and passion for the research project had a big impact on the thesis and motivated me to give it my all.

A special thanks goes to whole team of Baqend, namely Felix Gessert, Florian Bücklers, Hannes Kuhlmann and Malte Lauenroth, for their excellent technical support and the sublime work on their the BaaS system, in which I had the opportunity to integrate DCAT.

Furthermore, I would like to thank Prof. Dr.-Ing Norbert Ritter as well as Dr. Fabian Panse for supervising this thesis. Especially the feedback from Dr. Fabian Panse regarding expression, understandability and scientific correctness of the written thesis was tremendously helpful.

I would also like to thank Mirko Köster. Not only for his extremely helpful and detailed review of this thesis, but also for being a great partner and friend throughout my whole study.

Finally, I need to thank Sandra Kunz for her unconditional support, patience and inspiration.

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Des Weiteren erkläre ich, dass ich mit der Einstellung meiner Arbeit in die Bibliothek einverstanden bin.

Erik Witt

Hamburg, den 1. August 2016